TURING 图灵原创

- Oracle资深专家力作
- 以真实案例贯穿始终
- 感悟DBA思想精髓

DBA的思想天空

——感悟Oracle数据库本质

白 鳝 储学荣◎编著

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

TURING 图灵原创

DBA的思想天空

——感悟Oracle数据库本质

白 鳝 储学荣 编著

人民邮电出版社
北 京

图书在版编目（C I P）数据

DBA的思想天空 ： 感悟Oracle数据库本质 / 白鳝，
储学荣编著. -- 北京 ： 人民邮电出版社， 2012.10
（图灵原创）
ISBN 978-7-115-29443-2

Ⅰ. ①D… Ⅱ. ①白… ②储… Ⅲ. ①关系数据库－数
据库管理系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2012)第220663号

内 容 提 要

本书重在介绍Oracle数据库的性能调优方法及相应的工作思路，但并不拘泥于技术细节。作者结合多年的丰富经验，借助大量真实案例剖析了相关技术原理，阐述了理论知识在实践中的应用方法，并总结出“思路是道，操作方法是技。得道是极大的提升，也是DBA的思想精髓”的精辟论断。

全书分为三个部分，共19章。第一部分介绍了Oracle的基本原理，以及从基本原理衍生而出的一些分析问题的方法和思路。第二部分介绍了DBA应该掌握的常用工具。第三部分介绍了DBA分析问题的主要思路和一些典型案例。

图灵原创
DBA的思想天空
——感悟Oracle数据库本质

◆ 编　　著　白　鳝　储学荣
　责任编辑　王军花
　执行编辑　赵慧明

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
　邮编　100061　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　　印刷

◆ 开本：800×1000　1/16
　印张：28
　字数：668千字　　　　　2012年10月第1版
　印数：1－4 000册　　　2012年10月北京第1次印刷

ISBN 978-7-115-29443-2

定价：89.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

写在前言之前的话

按理说前言之前就不应该说什么了。不过这几天我真的有一种不吐不快的感觉，如果不马上把这些东西写出来和大家共享，就好像会对读者有所愧疚似的。今年 5 月 20 号是老白的母校南京大学的 110 周年庆典，也是我们南大计算机系 88 级同学毕业 20 周年，于是有好事之徒组织了一次同学聚会。遇到这种大碗喝酒、大块吃肉的好事，老白自然是积极响应的。不过既然是南大校友的聚会，当然会有所不同了。我们安排了一个活动，请当年最英俊潇洒、才华横溢的陈道蓄老师为我们再讲一次课，让我们回味一下 20 年前当学生的感觉。

本以为这就是一次活动而已，并没有十分认真。没想到这个活动最终成为“全国 60 名优秀中学生和诺贝尔奖获得者面对面活动”中的一个环节，安排我们和来自全国的 60 名优秀中学生一起聆听陈道蓄教授的讲座。而陈道蓄教授也一如 20 年前的严谨，为我们精心准备了一次 30 分钟的讲座，题目是《软件工程中的“软实力”》。这个和软件工程有关的讲座，以几幅世界名画导入软件工程中的核心问题，让人回味无穷。更令我感到惊喜的是陈老师关于“道”的感悟，他对于形而上的追索，让我有一种醍醐灌顶的感觉。由于音响调试得不好，坐在后排的我只能支棱着耳朵，认真地听他所讲述的每一个字，直到全场响起热烈的掌声，我才从沉醉之中醒来。我想，这 30 分钟，足以让我受益 10 年。而陈老师关于道的感悟，正和老白这本书的宗旨契合，悟道是“Thinking in Oracle”的最高境界。

陈老师谈到做一件事，总是会碰到“道”和“器”。道是形而上的东西，是别人无法教授的，只能靠自己感悟；器是有形的东西，很容易获得，也最容易使人从中受益。由于悟道艰难，因此大多数人都会追求器而放弃悟道。不过道终究是形而上的东西，一旦悟道，那么器就显得那么渺小了。

对于 DBA 也是如此，很多朋友整天都在追求某个独家秘籍，比如一个内部才有（Internal ONLY）的文档，或者一个脚本、工具，而如果你让他去认真读读 *Oracle Concepts*，他就会很不耐烦。前些天，公司的一个小伙子要推荐一个工具给老储，看老储没什么反应，就着急地跑过去和他解释这是个什么东西。老储也是个很直白的人，他说一般我们会根据原理来分析该如何处理，再去查看资料，看看具体的命令，平时没必要去记这些东西。我觉得老储这话说得很好，由道御器是再顺理成章不过的事情了。一旦你能够入道了，那么“形而下”的东西就太好办了。可能我并不知道某个著名的脚本，但是一旦我入了道，那么要做类似的分析，就很容易写出类似的脚本。而如果你并未掌握这个问题的相关原理，那么哪怕你拿着无数精妙的脚本，也可能分析不出什么东西。

很多 DBA 在学习过程中，总是会不断地追求些解决问题的具体办法，一本好书、一个脚本、一套工具软件，都会让我们感到兴奋。而实际上，这些都属于“器”的范畴，如果把它们作为我们悟道路上的一些工具和手段，那么是无可厚非的，不过你如果把追求这些当成自己追求的终极目标，那就舍本逐末了。我们不应该花太多的精力去追求这些东西，而应该把更多时间用在感悟 Oracle 本身的“道”上。公司的一个同事总是会发给我一些新奇的脚本或者一些不错的书籍，不过我一般都会拒绝。我会告诉他，你如果有时间就认真阅读一下，我没有时间看这些东西。这些脚本或者书籍都是很不错的，如果是 10 年前，我会很有兴趣认真地阅读。而现在，我最需要认真阅读的是 Oracle 的官方文档，从那里，我能够了解到最原汁原味的东西，从中感悟到更多的 Oracle 的本质。以我的理解，你如果已经掌握了某种方法，那么类似的方法哪怕有几十个上百个，大多数对你来说也是没有多大价值的了。如果你已经习惯于用某种工具去完成一个工作，那么其他的工具对你来说再好也是多余的。如果你总是在不停地学习一些新的工具，一些似是而非的新东西，而你就可能没有足够的时间去独立地思考和感悟。

因此，在 DBA 成长的过程中，除了如饥似渴地阅读，认真地参与实践之外，多花些时间来思考一些形而上的东西，对于突破瓶颈，尽快“入道”是很有帮助的。当你要去安装一套 11*g* RAC 的时候，你会怎么做呢？是马上去网上搜索一个攻略呢，还是认真阅读一遍 Oracle 官方的安装指南呢？我想绝大多数 DBA 会选择前者。根据攻略，你可以十分快地完成安装工作，而如果要去阅读安装指南，可能要花上一两天来认真思考和体会。这两者的效果区别是十分明显的，一个是别人悟出来的东西，你直接就可以拿来使用，很快，不用费心思；另外一条路是你自己去感悟一些东西，去解决一些问题，因此可能要花费更多的时间和脑力。不过，如果你真的通过认真阅读安装手册，经过多次实践，充分掌握了安装中的一些过程，那么你今后再做 11*g* RAC 安装的时候，哪怕碰到再多的困难，也可以自行解决，而采用第一种方法的朋友恐怕碰到一点点问题就只能再去谷歌或者别处打听了。

去年，一个网友打电话给我，咨询一个 11.2.0.2 GRID 的安装问题，我们排除了那个著名的思科交换机问题后，这个问题还是没有解决，后来我建议他跟踪一下 root.sh 这个脚本，搞清楚到底在什么地方有问题。最后这个问题终于找到了，是由于网卡不稳定导致的。不过通过这次安装，他彻底地分析了 root.sh 中的每个步骤的细节，他觉得今后再碰到 root.sh 出现的问题，就十分有信心了。

在学习过程中，不要过于依赖快餐式的攻略，多静下心来，认真思考和感悟一下 Oracle 的本质，对每个 DBA 来说都是十分有意义和有价值的。只有通过御器，从而入道，才是最佳的路径。这是陈老师的讲座给我的启示，我也希望我对此的感悟能够传递给每个读者。

白鳝
2012 年 5 月 19 日于南京

前　　言

写完《Oracle优化日记：一个金牌DBA的故事》和《ORACLE RAC日记》后，很多网友问我下面是否会继续写书。我的想法还是和以前一样，先整理整理自己的思路，写一些东西发在Oracle粉丝网上，写过一段时间再根据已经完成的内容决定新书的结构。开始我是想把新书写成DBA日记系列第三部的，不过写作的过程中才发现这种模式写下去有一点千篇一律的感觉了，很多案例从根本上看十分相似。有一次和同事老储聊天的时候，他提到现在很多年轻人不会按照Oracle的内在原理去考虑问题，从而导致经常出现常识性的错误。他的这句话就像火花一样在我脑海中闪现，Thinking in Oracle这几个英文词汇就出现在我的脑中。如果每个人都能以Oracle的基本原理为依据去考虑问题,那不是很好吗？老储的英文名字是John,在我的朋友里有两个John。一个是老外John，我还在玩ICQ的时候认识的网友，一个Oracle技术狂人。不过老外John现在已经成为一家银行的IT技术主管，随着岁月的流逝，当年的技术狂人现在已经成了狂热的人文主义者。去年圣诞新年假期他刚刚完成了10000公里行程的中南美洲自驾游，所到之处全部入住当地最高级的酒店，吃当地最昂贵的美食，还常常邂逅美女，虽然邮件还只是寥寥数字，但是羡慕嫉妒恨已经让我把这个资本主义的老家伙好好骂了几天。另外一个John就是老储，他属于睡在我对面的弟兄，我大学时的室友。自从1995年我把他从unixware汉化小组忽悠到深圳后，我们一直在一起合作。老储是个很低调的人，低调到第一次和客户打交道的时候客户可能会质疑他的能力，不过随着和他交往的深入，你可以发现他的深不可测。

老储的建议让我重新定义了本书的结构，把它分为基础知识、工具、方法和案例四大部分。不过随着写作的深入，我发现这个工作是十分艰巨的。这样一本书结构之大，内容之庞杂，远远超出了我的想象。于是我重新调整结构，将工具这一部分的内容从书中拿掉，准备独立成书，把方法和案例用两章讲完。这样调整后本书的内容更为紧凑，篇幅也可以控制在450页左右。

更大的惊喜也接踵而来，我终于说服了老储，让他参与到本书的写作中来。其实第一次和老储沟通这本书的时候我就邀请过他，不过被他以没空为由拒绝了。老储是一个懒散的人，我每次给他布置写文档的任务，他总是在快到期的时候才开始动手，不过每次提交的东西都让人无可挑剔。能让这么靠谱的人参与本书的写作，确实可以为书增色不少。老储凭着深厚的开发功力，在性能优化方面具有很敏锐的观察力，往往能够在很短很短的时间内找到系统的关键问题，而且他在Oracle文件结构和ASM的结构方面的研究很深，自己也编写了一个类似DUL的工具。有了他的加入，关于ASM原理和数据文件结构这部分内容我就可以推给他了。此前我正为这两节犯愁，考虑是否在书中去掉这两节，补充一些其他内容。老储的加入使这两节保留了下来，对于数

据文件结构和 ASM 文件结构感兴趣的朋友可能会感到庆幸，保留这两节也使得本书的内容更加丰富。

我给本书起的名字是《Thinking in ORACLE 之 DBA 的思想天空》，主编觉得这是一个十分霸气的名字。实际上，透过某个事物的本质去看问题，无论针对什么，都是比较高的境界。对于某些事情，在没有弄清楚其本质之前，我们往往难以找到正确的应对方法，虽然偶尔我们会像瞎猫碰到死耗子一样歪打正着，但是好运气不会总是伴随着你。就像前些年，我不了解“回南天”的成因，因此在每年的 2、3 月份总是十分痛苦。在广东沿海生活过的人都知道每年的春天总是会碰到回南天，时间有长有短，至少也在半个月左右。在回南天里，家里到处都是湿漉漉的，地上、墙上甚至天花板上都会渗出水珠。在这样的环境中生活半个月，绝对是十分恐怖的事情。为了让家里尽快干起来，我的第一反应是门窗大开，同时用风扇拼命扇风。不过这样处理并不能减轻返水的现象，有时候反而水更加多了。后来我请教了一位搞大气海洋研究的人，他告诉我回南天的成因是春天东南季风带来大量的水气，而当气温回升的时候，室外气温高于室内气温，湿热的空气遇到室内较冷的物体时，就会发生冷凝现象，从而就引发了反水的现象。一旦了解了回南天的成因，就很容易找到对付回南天的办法了，只要碰到气温大幅度回升的天气，就门窗紧闭。靠着这个办法，我终于摆脱了回南天的困扰，无论门外的走廊湿成什么样，我家的地上总是干干的。后来每当我看到朋友家里满地积水的时候，就会把这个方法教给他们，他们也逐渐远离了回南天的困扰。

从这个生活案例中，我们也可以看到，一旦了解了问题的本质，就很容易找到正确的解决方法。而没有理解问题本质的时候，我们所采取的应对措施不一定是靠谱的。

这个原则应用到 Oracle 数据库方面，也是一样的。对于每个来应聘 DBA 的人我都会问他们一个问题：“Oracle 到底是什么？”有些人会用数据库基础的理论来回答我：“数据库是数据的集合。”也有些人会感到茫然，不知道我问这个问题是什么意思。实际上很多 Oracle DBA 从来没有思考过这个问题。“Oracle 就是 Oracle，是一个产品，还能有什么意思呢？我不知道 Oracle 到底是什么也没有影响到我做一个合格的 DBA。”很多人都会这么想。

实际上对于 Oracle 我们确实还需要重新去认识认识，每个 DBA 在学习 Oracle 的时候都往往注重于学习如何建库、如何管理、如何编程、如何优化。我们总在不停地去学习一些方法，学习一些秘籍。如果偶尔学到了一些不传之秘，都会感到兴奋异常。也有些人使用这些秘籍解决了一些疑难杂症，成为了大家传说中的高手。

虽然说这也是学习 Oracle 数据库最为常见的一种方法，但是这样学习下去，我们总是在记忆一些枯燥的语法和脚本，虽然经过数年我们积累下了大量的经验，但还是无法真正地理解 Oracle，数据库升级了，系统变化了，我们就必须从头去学习。常年累月，我们总是在一次一次循环往复地重复着同样的事情，直到筋疲力尽，对 Oracle 失去往日的激情，最终 DBA 只是一个职业，Oracle 只是我们谋生的手段。这样学习下去，几年后，很多人就会碰到瓶颈，虽然说自己处理问题的能力和工作经验已经很丰富了，但是技术好像停滞不前了。我周围一些做了五六年 DBA 工作的朋友都遇到过类似的情况，他们咨询我的时候，我告诉他们，这是因为经验的积累已经到了一定程度，需要对 Oracle 基础概念有更深刻的认识。这种情况下，你需要静下心来认真看书，学习 Oracle

的基础概念，只有彻底搞清楚了这些，才能跨过这道坎，达到一个新的境界。绝大多数工作了五六年的朋友已经无法静下心来做这些事情了，因此他们失去了突破的机会。不过也没关系，大多数人选择了新的职业规划，从事管理，或者转向售前、业务专家等职位。

事实上，我们可以换一种方式来学习 Oracle，让 Oracle 的精神融入 DBA 的血液中，让 DBA 像 Oracle 一样思考问题，使 Oracle 成为我们的爱好，作为我们生命的一部分存在。对于大多数 DBA 来说，这也许只是一个乌托邦式的理想，多数 DBA 只是需要有一份工作，需要靠这份工作来生存，娶妻生子，享受生活。并不是所有的人都希望让 Oracle 成为自己生命的一部分，这是很现实的，不过我们虽然可以仅仅把 Oracle 当做是谋生手段，但也还是可以同时尝试了解 Oracle 更多的本质，像 Oracle 一样思考。

对大多数人而言，像 Oracle 一样思考虽然不能带给你更多的生活乐趣，但是通过以这样的方式去学习和思考，会更加精确地了解 Oracle 的精髓，让自己在 DBA 的成长过程中少走弯路。10 多年前我第一次接触 Java 的时候，感到十分头痛。不是自夸，10 多年前，我是一个相当不错的 C 程序员，最高纪录是一天之内编写 500 多行复杂的代码，而且一次性编译通过，一次性测试通过，这样的记录的诞生是基于十分良好的过程思维能力的。不过我这个自封的编程高手第一次接触 Java 的时候，却感到十分吃力。我无法用面向对象的思想去编写程序，所以学起 Java 来十分痛苦，几次学习，最后都放弃了。直到有一天我看到了一本英文的书籍 *Thinking in Java*，通过这本书，我掌握了 Java 和面向对象设计、编程的主要思路。自从看了这本书之后，我再次面对 Java 程序的时候，发现一切都是那么地简单，很快我就掌握了 Java 编程。现在我虽仍然还只是一个三流的 Java 程序员，不过粉丝网的一些修修补补的工作我完全能够胜任了，而且在和一些 Java 开发人员交流的时候，我也能够很快地理解他们的思路。

后来我总结了一下，在看 *Thinking in Java* 这本书之前，我在编写 Java 程序的时候，并没有理解面向对象编程的概念，只能是照猫画虎，拿着一个例子在上面修改。实际上我的编程风格还是面向过程的，因此写出来的代码质量很差。而通过阅读 *Thinking in Java*，我终于学会了面向对象的方法，用 Java 本身的思想去考虑问题，因此能够更加准确地抓住问题的本质。我想，学习 Oracle 数据库也是这样，如果我们通过一个又一个案例去学习 Oracle，那么将永远停留在表层上。有些 DBA 只能重复相同的案例，这样的 DBA，哪怕干上 10 年 20 年，也可能只学到 Oracle 的一些皮毛，碰到一个没有见到的案例，可能就会感到手足无措。而水平高一些的 DBA 往往能够判断出案例的相似性，并通过分析找到类似案例的解决方法，这其实就是因为透过现象看到了问题的本质。

很多 DBA 可能都碰到过我下面所说的一些问题，有时候我们无法评估某项调整可能对系统带来的影响，有时候我们面对一个复杂局面的时候很难快速找到问题的关键，也有时候我们在为解决某种等待事件而感到无从入手。实际上，遇到这些问题，都是因为缺乏对于 Oracle 内部原理的充分认识。在很多情况下，当经验无法为我们提供足够支持的时候，就必须从原理出发进行思考，才有可能真正掌握问题的根源，从而解决问题。

前几天我在一个客户现场做数据拯救工作的时候，他们的备库（也就是现在的主生产库）突然宕机了，当时客户正在做一个删除临时文件的操作，领导就认为他这个操作导致了宕机，而操

作人员也觉得很冤枉，因为这是一个十分常规的操作。我看了看日志文件，从日志上看不出任何由于临时表空间和临时段操作引起的问题，同时又看到了一个好像是人工操作停库的信息，于是推断可能是人为操作所致。后来经过多方面查证，确实是有个 DBA 在家里远程做维护的时候，发现操作 HANG 住了，情急之下，直接重启了数据库。如果你不了解临时段和临时表空间操作的原理，面对这个问题，很可能一上来就把重点放在删除临时文件导致宕机问题的分析上面，这样就偏离了正确的方向，解决问题的效率和成功率就会大大降低。

我们强调理论的重要性，也不是片面强调理论而不重视实践。Oracle 数据库是实践性很强的，没有实践，光学习理论是无法成为真正的高手的。比如说我们学习了很多 OWI 相关的理论，了解了数据库等待事件和一些状态指标的含义，但是我们看到一个库的 AWR 报告的时候，还是无法知道某个指标是否正常。当对大型 OLTP 系统缺乏实践经验的时候，我们就无法知道大型 OLTP 系统的一些技术指标的特性，因此也很难从中找到疑点，进而找到解决问题的方法。

这些年里我接触过大量的 DBA，我一般把他们分为四大类。第一类 DBA 是经验型的，处理问题的主要方式取决于以往的经验，他们往往都有很好的习惯，会把每一个处理过的案例整理出来，今后再碰到这类案例的时候，他们会很快地解决问题。随着工作时间的增长，他们的技术也会相应地提高。第二类 DBA 是理论型的，他们具有很深的理论基础，经常探讨一些“Oracle Internal Only”的高深问题，比如他们能够很清晰地告诉你共享池分配的算法，告诉你 checkpoint 的工作原理，但是这些 DBA 往往缺乏实际的工作经验，他们研究 Oracle 却很少有机会接触大型的数据库系统，因此实际解决问题的能力并不强。另外，由于他们的知识比较片面，在某些方面很深入，而某些方面就是浅尝辄止，这种不均衡导致他们的知识只是以点的形式存在，无法串成整体，因此那些很深入的研究并不能给他们实际工作带来多大的帮助。第三类 DBA 是技巧型的，他们并不注重理论的学习和经验的积累，在处理问题的时候往往能够利用 Metalink 和谷歌、百度之类的工具去搜索解决方案，这类 DBA 最为常见。他们处理问题往往靠运气，而且一些他们自鸣得意的成功案例往往也是经不起推敲的，下回碰到类似案例时，可能还会失败。第四类 DBA 是虚心请教型的，他们无论碰到什么问题，甚至连错误信息都没有看明白，就开始叫“我的系统出问题了”，然后到处去问如何解决。

实际上，这四类 DBA 都是有缺陷的。第一类 DBA 可能经过多年的工作，有十分丰富的经验，处理问题的能力很强，而且分析问题十分敏感，很容易抓到问题的关键，但是由于没有深入理解 Oracle 的理论，碰到一些较为深入的问题的时候，就不容易立刻找到关键。虽然凭借着自身丰富的经验和问题分析排查能力，他们最终也能解决大部分的问题，但是往往问题解决后还是没有真正弄明白为什么会解决问题，下一次碰到类似的问题，可能还是要花很大的代价。

第二类 DBA 在某些方面的理论知识很强，总是喜欢研究一些十分高深的原理性的东西，但是这类 DBA 的主要精力都放在了研究一些 Oracle 内部原理上了，他们没有很多的时间去实践他们学到的理论。这类 DBA 往往知识面较为狭窄，仅精通于自己研究比较深入的领域，在实际工作中也很难发挥出自身的理论研究特长。

第三类 DBA 实际上在我们的现实生活中是最常见的，“万事不明问百度，百度不明就抓瞎”，确实谷歌、百度和 Metalink 能够帮助我们解决不少问题，但是这类 DBA 往往在问题解决后没有

好好思考一下，为什么这个方法能够解决问题，更没有认真总结和归纳一下，下一次碰到类似的问题，还是无法依靠自己的思考去解决问题。于是再 Google 一把，也许这一次运气没有那么好了，Google 出来的资料不是上回的那个了，于是结果可能是很悲惨的。

第四类 DBA 在我们现实生活中也经常出现，网络社会通信十分发达，打个电话或者在 qq 群里、msn 里问问，也许就有人帮忙解决问题。久而久之，这些人放弃了自己的思考，碰到一点点小问题都要找人问。缺乏独立思考问题能力的 DBA，只能称为一个数据库操作员，实际上离真正的 DBA 还有十万八千里呢。

看到这里，大家可能明白了，老白实际上说的不是四类 DBA，而是 DBA 的四种性格，这四种性格可能会集中在某一个人身上。以老白学习 DBA 的经验来看，理论结合实践是十分重要的。在 2000 年前，老白虽然做了很多项目，也是很多人眼里的 Oracle 数据库高手，但那时的老白就是第一类 DBA 的典型，没有经过多少理论学习，几乎所有的 Oracle 数据库的技能都是从实践中获得的。虽然在实践中我总结出大量的经验，甚至有很多客户建议我写一本书，把我对 Oracle 的理解写出来，不过当我自信满满地开始写书的时候，却突然发现，我的一些知识需要进行确认，否则写出来就贻笑大方了。于是我开始大量地学习 Oracle 的一些理论知识，随着写书过程的深入，我越发感到自身理论水平的不足。《Oracle 数据库深度历险》这本书我写了 3 年，实际上 2002 年就彻底放弃了出版这本书的念头，因为我发现自己的理论知识确实还需要进一步的梳理。但是我并没有放弃写作，因为我发现通过写作，我更为系统地将 Oracle 的理论知识梳理了一遍，这次梳理是通过我以前的知识体系、工作经验，与 *Oracle Concepts* 的理论基础进行了一次完整的整合。通过这 3 年的写作，我终于完全疏通了自己的 Oracle 的理论体系，好像一个练武术的人，终于打通了任督二脉，感到无比的畅快。

听老白说了这么一大通，是不是很多人都感觉到手脚发凉，难道成为一个合格的 DBA 有这么难吗？如果我没有打通任督二脉，就不算一个合格的 DBA 吗？实际上 DBA 成长的道路是很多的，并不一定要走老白这一条路，老白仅仅是根据自身的经历，通过这本书来帮助大家梳理 Oracle 的一些基础知识而已。还是那句话，如果 Oracle 是你的爱好，那么你无论花多大代价去研究它都是值得的；如果 Oracle 只是你职场生涯中的一份工作而已，那么只要你认真对待它就可以了，没必要像老白那样执著。

作为一个 DBA，理论学习和实践如何相结合是十分关键的。在初期，一般来说 DBA 都是通过了某种途径接触了 Oracle 数据库，进行了一系列的操作。在工作过程中发现了一些问题，才开始想到需要去看一些 Oracle 的书籍。在这个阶段，Oracle 官方文档的 2days、7 days 系列入门书籍就十分有效。通过这些书籍你可以了解 Oracle 的一些基本的原理和基本的操作，帮你在工作中解决部分问题。这样你在工作中就能够应对一些简单的问题了。不过碰到稍微复杂一些的情况，你可能还是会发懵，这时，*Oracle Concepts* 这本书就十分关键了。从这个阶段开始看这本书是十分必要的，它有助于你在积累经验的过程中不断地完善理论。不过，你可能还无法完全理解 *Oracle Concepts* 中的基本概念，通读这本书是十分必要的，但是不必要把每个问题都搞得十分清楚。因为要达到这一点，你需要花费太多的时间和精力，同时也可能会由于缺乏足够的技术指导而无法真正理解问题的本质。不过在这个阶段，碰到某些问题或者研究各种案例的时候，经常翻翻 *Oracle*

Concepts 这本书是十分有益的，因为在处理问题的时候，你针对这个问题的思考会比较深入，这个时候，认真分析一下相关的理论是十分有效的。对于处理过的每一件事情，都做一个比较详细的记录，是十分好的习惯。记录下某个案例，可以供水平提高后再进行回顾，或者将案例提交给某个专家去评审，或者在网上和大家一起讨论，对于学习 Oracle 的原理都是十分好的方法，可以帮助你在分析案例时提高对 Oracle 数据库原理的认知。

在这本书里，老白会把《Oracle 数据库深度历险》中的一些内容，结合老白的实际工作经验展现给大家。我会剖析原理，并结合案例来说明这些理论知识如何在实践中应用。希望老白的这次写作经历，能够给大家带来一些帮助。

《DBA 的思想天空》二次印刷了，在本书出版后，大量读者对本书提出了勘误。老白在此对大家表示感谢，只有认真的读者，才能成就高质量的图书。图灵的书一直是老白比较喜欢的，因为图灵对图书质量的把控一直很严。虽然作者和编辑已经尽可能认真，但是本书是老白在将近一年的时间里完成的，其中难免错漏，甚至有些观点也不尽正确。老白希望在和大家分享自己的思想心得的同时，大家也能帮助老白发现书中的问题。有些问题甚至可以通过激烈的讨论，道理是越辩越明的。其实在本书刚开始写作时，老白引用了一个共享池故障的处理案例，不过由于后来和网友讨论这个案例的时候，发现有一些支撑某个观点的依据并不充分，因此这个十分有趣的案例在最终成书时被舍弃了。因为这一点不明确，可能导致老白要表达的某个观点无法得到支撑，甚至会误导读者。

老白会继续通过写作来和大家分享自己对 Oracle 的感悟，也十分希望能继续和大家互动，希望大家继续为老白的书提勘误，让每一个印次的质量都能有所提高。还是那句话，只有认真的读者才能成就高质量的图书，与大家共勉。

阅读本书的建议

本书还是一本介绍方法的书，并不是一本系统介绍 Oracle 知识的百科全书，因此读者在阅读本书时应该注意方式方法，需要注重对基础概念的理解和对工作思路的理解，而不要拘泥于某个技术细节。本书并没有涉及任何技术细节，关于技术细节，读者可以参考 Oracle 相关的官方文档，比如 *Complete Reference*、*Oracle Concepts*、*Oracle Performance Tuning Guide* 等。实际上，老白认为 Oracle 的官方性能优化手册是我看到过的最好的性能优化方面的书籍，没有之一。如果你认真研读过这本技术手册，就会发现你以前看到过的关于 Oracle 性能调优的书籍的核心内容，只不过是对这本书的另外一种阐述而已，可能会增加一些例子，可能在某些细节上会更加细致，仅此而已。

本书中的一些观点仅仅是老白自己这些年对 Oracle 的理解，并不是金科玉律，可能有一些思路和方法是有一定局限性的，甚至有些或许是错误的，因此读者不能盲从。结合自己的知识体系，重新认识这些思路和概念，使之成为自己的知识体系的一部分，并用于指导你建立自己的问题分析、预案处理体系，才是最为根本的。如果某些案例或者某些观点你不太认同，欢迎大家到 www.oraclefans.cn 上去和老白探讨。有互动的阅读，可能会对你的帮助更大，也有助于老白纠正自己的一些错误。

本书中的一些例子都是比较容易去实践的，老白其实仅仅使用了一套 Oracle 10*g* 的数据库、一个带 sql*plus 的 Oracle 客户端，再加上 profiler 工具，就完成了本书中绝大多数实验。希望有兴趣的朋友可以亲手去做一做这里的实验。如果大家觉得里面的脚本自己敲出来很麻烦，可以到 www.oraclefans.cn 上发帖向老白索取。不过老白觉得，如果你能看懂这些脚本，并自己再写出来，那么这些脚本就真正成为了你自己的工具。而从网站上下载下来的工具，可能很快就会被你丢弃在一边，过几天就忘记了。

目　　录

Part 1 第一部分

基础原理篇

实际上对于 Oracle 我们确实还需要重新去认识。很多朋友都会游泳，那么我来问一个简单的问题：什么叫做真正地学会了游泳？每个学游泳的人都有这样的感受，刚刚学习游泳的时候最大的问题是无法浮在水面上；经过努力，发现只要手脚按照一定方式滑动，身体放平就能浮在水面上了，这个时候就感觉从不会游泳变成会游泳了。但是这时候你会面临另外一个问题，就是不会换气，学会换气后再往后学习就容易了很多，不过可能我们游上几十米、百把米就会觉得很累。后来我们发现，原来我们游泳的姿势还不太标准，而如果我们学会了踩水，游多远都不是个问题了，这个时候我们就如鱼得水了。学习 Oracle 其实和学习游泳十分类似，刚刚开始的时候，我们处于入门阶段，要克服对 Oracle 的恐惧心理，只要我们想学好 Oracle，我们就一定能做到，实际上 Oracle 在入门阶段是十分容易的。我们从头学习 Oracle，学会了安装数据库，学会了管理表空间，学会了日常的故障处理。就像我们学游泳的时候刚刚学会换气一样，虽然感觉已经掌握了 Oracle 数据库，但是碰到稍微复杂一些的问题我们还是觉得很难入手。我们能看清楚一些东西，但是感觉还是看不远。

上面这种情况是每个初学者都会碰到的，学 Oracle 和学游泳一样，刚开始的时候我们只是学会了游泳的外部表现，学会了像别人一样划水、换气，但是并没有真正掌握游泳的本质。如果要消除初学期的迷茫，我们一定要真正地掌握 Oracle 数据库的概念，一个 DBA 如果连认真学习 Oracle 的基本概念都不能做到，那么我觉得他也很难成为一个真正的高手。

这些年我在网上和大家一起讨论 Oracle 的问题，也经常有朋友问我，要成为一个优秀的 DBA，应该看些什么书。我给大家推荐的第一本书就是 *Oracle Concepts*，这也是我唯一能够推荐给所有人的书。这本书确实适合任何一个 DBA 阅读，无论你是初学者还是已经工作了很多年的高手。只要你没有认真读过这本书，你就有必要去读一遍。

理解 Oracle 的基本原理有助于你用一种十分理性的思维去考虑问题，在处理问题的时候能够更快地抓住问题的本质。我们大家在从事 DBA 工作的时候，经常会碰到这样的情形，一个问题困扰了我们很长时间，突然猛地一下，你就抓住了问题的关键，然后它就迎刃而解了。而在这之前，我们可能会做过很多尝试，这些尝试有些是有序的，有些是无序的，甚至有些只是瞎蒙。在这种情况下，有时候经验是帮助不了你的，因为所有根据经验的分析都已经做完了，并且被证明是无效的。这时，我们就只能根据自己对问题本质的理解去分析，才可能最终解决问题。

事实上，要想像 Oracle 一样思考，首先就需要了解 Oracle 的本质是什么，它是怎么运作的，在运作过程中，哪些地方可能成为瓶颈。这个时候可能就有朋友感到疑惑了：我们如何了解 Oracle 内部运作的原理呢？这看似一个不可能完成的任务。确实，在处理有些问题的时候，我们可能要了解 Oracle 很深入的算法，但是在绝大多数情况下，我们分析问题只需要了解一些我们在 Oracle 官方文档中提及的原理性的东西，并不需要深入到算法和源代码级别。比如说，如果碰到一个共享池的性能问题，你会马上联系到共享池、库缓存（library cache）、字典缓存（row cache）、CURSOR，以及相关的一些闩锁、参数等，在大多数情况下这些就可以支撑你做分析了。但是必须把这些知识点编织为一张网，而不是一个一个的知识点。对于大多数 DBA 来说，想把这些知识点编织为一张网是比较难的。这需要你花费大量的时间来学习，而很少有一本书能够从这个角度去介绍知识。因此这种学习会比较困难。在本章，老白将尝试以这种方式来介绍相关的知识点，帮助大家从知识点升级为一个立体的知识体系。

另外，把知识点融会贯通不是看书就能达到的，要想达到这个目标，必须进行足够的实践活动。你只有在把学到的知识应用在实践中，才可能突破知识孤岛，达到新的境界。因此，如果你看完第一部分内容后，有些知识点还感觉无法掌握，这也并不要紧，结合自己碰到的案例思考一下，也许会更有收获。

第一篇的大多数章节都是用“理解”开头，因为这部分是以理解 Oracle 基本原理为主要宗旨的章节。通过本篇，老白希望大家能够对以前有一定认识的 Oracle 认识得更为深入，那么你买这本书就不冤枉了。

第 1 章 理解 Oracle 数据库和实例

数据库和实例是 Oracle 最为基础的两个概念，只有很好地理解了什么是数据库，什么是数据库实例，才能更好地理解 Oracle 的其他概念。本章将会通过一些知识点来了解什么是数据库和数据库实例。通过本章的学习，我们也会更加适应本书的风格。

1.1 什么是 Oracle 数据库

什么是 Oracle 数据库呢？这是一个我经常问 DBA 的问题，今天我要来回答这个问题。说到数据库，它是数据的集合，这一点就没必要在这里详细讨论了。大多数介绍数据库原理和数据库基础的教材要比我解释的专业得多。这里只讨论“什么是 Oracle 数据库”。Oracle 数据库是甲骨文公司开发的一种关系型数据库管理系统，也就是 RDBMS。在这里我必须插几句，说说我对拉里·埃里森的崇拜之情。我曾经崇拜过乔布斯，不过那是我对 80 年代发明苹果电脑的乔布斯的崇拜，也许 iPhone 是乔布斯人生辉煌的顶点，但是我只崇拜发明了那台绿色字符的小电脑的乔布斯；我也曾崇拜比尔·盖茨，不过那是我对 DOS 3.0 的崇拜。但自从听说了拉里用锤子为办公室开辟网线通道的故事（不管这个故事是不是真实的），我就开始崇拜他了。用圣迹来命名一家公司和一个产品，这不是我们这种凡夫俗子能够做到的。Oracle 也确实像圣迹一样，深深地影响着全世界。

Oracle RDBMS 是一款十分优秀的关系型数据库产品，Oracle 从头到尾都是一个 RDBMS，是针对 OLTP 系统进行设计的，这一点从它底层的块结构就可以看出。Oracle 在大并发量和海量数据关系型检索方面具有十分优越的性能，但是它并不擅长 OLAP，因为它不支持列压缩存储（当然，从 exadata 开始，Oracle 也能够支持混合列压缩，这是一种行存储和列压缩的混合模式，目前只在 exadata 数据库一体机上实现）。与其他关系型数据库相比，Oracle 在某些方面更为优秀，但也有其不足的地方，因此它绝对不是万能的。优势和劣势都是与生俱来的，这是由 Oracle 数据库的基本架构和数据存储的基础结构所决定的，优化只能解决局部性的问题，有限度地提升其性能，但是绝对无法完全掩盖结构性问题带来的负面影响。Oracle 的优势在于大并发量下的高吞吐能力，因此很适合大型企业级应用。但是如果我们在一个并发量和数据量都不是很大的系统中，对 Oracle 和 MS SQL Server 进行比较，就不难发现 Oracle 并没有多大的优势，甚至在某些方面还不如后者。

再往更为本质的方面去探讨，Oracle 是一个 RDBMS 系统，也是一款应用软件。Oracle 数据库除了将数据存储于文件中外，还通过一个被称为实例的后台机制向外提供服务，这两部分我们将在随后的两节中详细介绍，这里不做过多的描述。我们在这里仅仅讨论作为应用程序的 Oracle RDBMS 系统，它必须依赖于某个操作系统或硬件体系。和我们自己编写的程序一样，Oracle 必须适应于某个操作系统，并充分利用操作系统提供的资源，反过来，操作系统也必须能够将资源提供给 Oracle 数据库使用。在一个仅仅运行 Oracle RDBMS 的系统上，操作系统应该被调整为能够将绝大多数的资源都提供给 Oracle 数据库。这样，Oracle 的进程就能够最大可能地得到足够的系统资源。

讨论这个似乎又有点跑题了，其实不然，只有充分了解 Oracle 的本质，我们才不会神化 Oracle。Oracle 的本质就是一款软件、一个程序，那么它就具备程序的一切特征，包括可能出现的 bug。

但是 Oracle 不是一个简单的程序，而是十分复杂的体系。首先，Oracle 需要将数据存储在数据文件中，为了能够支持大量的并发用户访问数据库，并且提高数据库的访问性能，Oracle 需要引入共享内存，从而实现资源的共享。比如，针对 SQL 引擎，每个 SQL 最终将会被解析为一系列的执行步骤，这就是我们常说的执行计划。如果同一个 SQL 执行多次，每次都要重新生成执行计划，那么效率就比较低下了，Oracle 引入了共享池来实现这方面的共享。同样，如果一个数据块每次读取都要访问文件，那么效率就不高了，于是 Oracle 引入了 DB Cache 来缓存这些数据。同一个数据块可能被多个用户修改，如果每次修改就要直接存盘，那么效率也会降低，于是 Oracle 设计了 DBWR 进程，来专门负责将数据块写入文件。这似乎很复杂，不过这一切对于架构师来说很好理解。架构设计的目的就是有效地将功能划分成不同的组成部分，然后让这些部分能够很好地协同工作，从而达到最好的效果，因此架构师很容易做出类似的设计。

实际上，作为一个应用程序的 Oracle，它的实现原理是十分朴实的，并不像我们想象的那么神秘。前几天我碰到一个案例，有个客户的 Oracle 9.2.0.6 数据库突然出现了故障，sqlplus 通过 sysdba 能够登录，并且能够访问一些系统视图，比如 v$session，但如果使用普通用户登录就会被挂起。通过 HANGANALYZE 工具分析，没有发现任何异常。然而，在检查 Oracle 后台进程的时候，我们发现所有的 Oracle 后台进程和绝大多数前台进程都消失了。客户很是不解，为什么会这样呢？检查日志，没有发现任何异常。于是我们使用 shutdown abort 关闭了实例，并且进行了重启。我们都觉得没有日志，很难分析，也就没有深入研究。

第二天，客户又找到了我，说数据库又出现了昨天的情况，这回所有的 Oracle 进程，包括前台进程、后台进程，统统没有了，而且没有任何的日志产生。我考虑了半天，突然有所感悟，Oracle 的实例实际上也是一款应用软件，由多个进程组成，任何一个进程发现系统存在异常，都会第一时间记录日志，如果问题十分严重，就会关闭实例。我们可以使用 sysdba 账号登录系统，并且能够在 HANGANALYZE 工具中看到会话的信息，说明 Oracle 的共享内存还存在，只是所有的进程都没有了。这种情况只有一种可能，就是所有的 Oracle 后台进程都是在同一个时间点被终止的，而且不是程序自己退出的（因为程序自己退出，应该有机会完成自己的退出业务逻辑，比如写日志记录故障），而是被外力强行终止的。

从上面的分析，很自然就能联想到 Oracle 的后台进程很可能是被人为杀掉了。于是我做了一个实验，发现如果杀掉所有的后台进程，Oracle 的共享内存还是存在的，并且能够通过 sysdba 账号访问，普通用户登录由于缺乏后台进程的支持，会被挂起。这个现象和客户目前碰到的问题十分相似。

发现这个问题的真相只有一个渠道，就是从应用程序本质上去考虑，这样才能得出所有的后台进程都是在同一时间被终止的结论，并找出其原因。只有外部力量的介入，才有可能出现所有后台进程全部终止，而共享内存还保持正常的现象，这绝对不是某个 bug 能产生的结果。排除了 bug 的影响，我们才能把主要精力集中在正确的方向上。

到这里，对于这个案例的分析就接近尾声了，本节并没有很深入地介绍“什么是 Oracle”，而是更加直接地介绍了 Oracle 的本质。Oracle 在本质上就是一组应用软件，它也具备所有应用软件所具备的特征。了解这一点，是我们今后解决任何问题的基础。任何看似妖异的现象，都离不开 Oracle 作为应用软件的本质，都无法违背应用软件所遵循的规律。

作为应用程序的 Oracle，必须依赖于其运行的系统环境，Oracle 数据库的处理能力和性能也依赖于主机硬件、存储、网络和操作系统等因素，因此作为 DBA 不能仅仅就 Oracle 而论 Oracle，还必须熟悉 Oracle 运行所依赖的环境。作为应用程序的 Oracle，会和操作系统中的其他进程竞争有限的系统资源，因此，在数据库服务器上做一些比较大的操作时，一定要谨慎，因为这些操作可能会使 Oracle 数据库出现问题。

Oracle 不仅是特殊的应用程序，更是庞大的数据库管理系统，它包含了一个 RDBMS 管理系统和其他一系列应用程序。Oracle 的核心——RDBMS 管理系统包含在$ORACLE_HOME/bin/oracle 映像、$ORACLE_HOME/lib/libclntsh.so 等中，而 sqlplus、exp 等则是一些 Oracle 数据库的工具。tnslsnr 是 Oracle 的网络连接部件，用于连接客户端到 RDBMS。这些应用程序都被安装在 ORACLE HOME 目录下。

通过上述应用程序，RDBMS 管理系统及其工具，用户就可以创建、管理数据库。另外，用户还可以通过 sqlplus 工具，使用 create database 命令去创建一个数据库，也可以使用 startup 和 shutdown 命令去启动和关闭数据库。

数据库是独立的，从物理结构上看，它是由一系列文件组成的，包括参数文件、口令文件、控制文件、数据文件、日志文件等。一套完整的数据库，只要其所有的文件都是完整的，那么即使数据库的 RDBMS 管理系统遭到破坏，只要重新安装和数据库版本一致的 RDBMS 管理系统，该数据库就可以重新启用。其实所谓重新启用，本质上就是可以在某个实例中打开这个数据库，供客户使用。

另外需要注意的是，Oracle 数据库是一个 RDBMS 管理系统，其本质是关系型数据库。关系型数据库是十分适合 OLTP 应用的，因为它存储的是一系列的关系，各种关系以表的形式被存储起来。比如，春节前铁路网上售票系统崩溃，有人分析这是由于铁路系统固步自封，没有使用国外某大厂商的产品，而选用了通用 RDBMS 数据库产品所导致的性能问题，如果选用了某国际知名厂商的网状数据库，就不会有问题了。这个说法看似有理，实际上，如果足够了解 RDBMS，就知道其不值一驳了，因为铁路售票系统是十分典型的 OLTP 应用。

另外,由于 Oracle 是行存储的 RDBMS 数据库,这一特点也使其十分适合 OLTP 应用。从 Oracle 数据库的内部数据结构可以看出，Oracle 在行存储数据方面下足了功夫，甚至连行锁都是设置在行头中的。在行锁的性能方面，Oracle 的表现极为优秀，这一点毋庸置疑。不过这种设计，可能不适合一些经常以列为访问对象的 OLAP 系统，列压缩技术才是实现这类应用的最佳解决方案。因此如果有人宣称，Oracle 数据库在 OLAP 分析方面的性能高于 SYBASE IQ，那一定不可信。

1.2　Oracle 数据库的物理结构

Oracle 数据库的物理结构从本质上来说就是一系列的文件。Oracle 数据库分为几个部分，第一部分是 Oracle RDBMS 系统的安装目录,也就是我们常说的 ORACLE_HOME。ORACLE_HOME 包含了 Oracle 运行包的几乎所有的文件，当对 ORACLE_HOME 执行 tar 命令，并将其复制到一台具有相同操作系统的机器上后，解开包配置一些环境变量就可以使用了，一般来说都不需要做什么特殊的处理。不过由于我们安装操作系统时可能不会完全一致（操作系统的小版本、补丁包、安装的可选包等），因此针对通过 tar 命令复制过来的介质，应在使用前做一次重新链接。Oracle 在$ORACLE_HOME/bin 目录下，提供了用于重新链接的工具，只要进入该目录，执行：

```
$relink all
```

就可以完成 Oracle 介质的重新链接。当然为了便于今后管理，如果要复制一套 Oracle RDBMS 软件介质，不能仅仅复制 ORACLE_HOME，还需要创建 bdump、udump 等目录，为 Oracle 的前台和后台进程输出日志使用。除此之外，还有一个十分重要的 Oracle 组件需要进行复制，这就是 Inventory。

Inventory 是什么呢？很多做了多年维护工作的 DBA，可能仍然不太了解它。下面老白就和大家一起讨论 Inventory。

1.2.1　Inventory

什么是 Inventory 呢？Inventory 是 Oracle 安装工具 OUI 用来管理 Oracle 安装目录的。Inventory 里注册了某个 ORACLE_HOME 下安装的数据库的组件及其版本。Oracle 数据库软件的升级、增删组件，都需要使用 Inventory。在一台服务器上，Oracle OUI 会创建一个全局的 Inventory，全局 Inventory 的目录在 oraInst.loc 文件中指定。根据操作系统的不同，oraInst.loc 所在的目录也不一样。在 AIX 或者 LINUX 等系统中，oraInst.loc 存放在/etc 目录下，在有些操作系统中，这个文件存放在/var/opt/oracle 目录下。oraInst.loc 文件中包含下面的配置项目：

```
inventory_loc=<oraInventory 所在目录>
inst_group=<OUI 安装 ORACLE 的操作系统组>
```

比如说，老白的一台测试机上的 oraInst.loc 内容是这样的：

```
inventory_loc=/opt/oracle/oraInventory
inst_group=oinstall
```

这里面有两个信息，第一个是 Inventory 所在的目录位置，第二个是安装 Oracle 的组的名称。

这个 inst_group 参数十分重要，它会在 link Oracle 映像的时候被使用，如果这个参数设置错了，那么 link 出来的 Oracle 映像就无法被正常使用了。

在全局 Inventory 中定义了所有 Oracle HOME 的情况，这个文件就是 ContentsXML 目录下的 Inventory.xml：

```
<?xml version="1.0" standalone="yes" ?>
<!-- Copyright (c) 2001 Oracle Corporation. All rights Reserved -->
<!-- Do not modify the contents of this file by hand. -->
<Inventory>
<VERSION_INFO>
   <SAVED_WITH>2.2.0.18.0</SAVED_WITH>
   <MINIMUM_VER>2.1.0.6.0</MINIMUM_VER>
</VERSION_INFO>
<HOME_LIST>
<HOME NAME="Ora10gHome1" LOC="/opt/oracle/product/10g" TYPE="O" IDX="1"/>
<HOME NAME="ora9i" LOC="/home/ora9i/product/9204" TYPE="O" IDX="2"/>
</HOME_LIST>
</Inventory>
```

比如上面的例子中，老白的系统上有两个 Oracle Home，一个是 9*i* 的一个是 10*g* 的。

在 ORACLE_HOME 下面也有一个 Inventory 目录，这个目录就是我们平时说的 Local Inventory。这个 Inventory 是本地的，每个 ORACLE_HOME 所独有的。它记录了本 ORACLE_HOME 中 OUI 安装的组件的信息。

了解了 Inventory 的一些基础知识，下面我们就要聊聊了解这些知识有什么好处了。假设这么一个场景，我们去给一个客户的数据库打补丁，到了客户现场，首先使用 opatch lsInventory 来查看一下当前系统的情况。正常情况显示的数据是这样的：

```
Invoking OPatch 10.2.0.4.2
Oracle Interim Patch Installer version 10.2.0.4.2
Copyright (c) 2007, Oracle Corporation.  All rights reserved.
Oracle Home       : /opt/oracle/product/10g
Central Inventory : /opt/oracle/oraInventory
from              : /etc/oraInst.loc
OPatch version    : 10.2.0.4.2
OUI version       : 10.2.0.4.0
OUI location      : /opt/oracle/product/10g/oui
Log file location : /opt/oracle/product/10g/cfgtoollogs/opatch/opatch2011-05-06_11
-41-25AM.log
LsInventory Output file location : /opt/oracle/product/10g/cfgtoollogs/opatch/lsinv/
lsInventory2011-05-06_11-41-25AM.txt
--------------------------------------------------------------------------------
Installed Top-level Products (4):
Oracle Database 10g                                                   10.2.0.1.0
Oracle Database 10g Products                                          10.2.0.1.0
Oracle Database 10g Release 2 Patch Set 3                             10.2.0.4.0
Oracle Database Vault                                                 10.2.0.4.0
There are 4 products installed in this Oracle Home.

There are no Interim patches installed in this Oracle Home.
```

而不幸的是，我们看到了这样的信息：

```
[oracle@ OPatch]$ ./opatch lsInventory
Invoking OPatch 10.2.0.4.2
Oracle Interim Patch Installer version 10.2.0.4.2
Copyright (c) 2007, Oracle Corporation.  All rights reserved.
Oracle Home       : /opt/oracle/product/10g
Central Inventory : n/a
from              :
OPatch version    : 10.2.0.4.2
OUI version       : 10.2.0.4.0
OUI location      : /opt/oracle/product/10g/oui
Log file location : n/a
OPatch cannot find a valid oraInst.loc file to locate Central Inventory.
OPatch failed with error code 104
```

和客户沟通了以后我们发现，这套系统不是安装的，而是开发商直接 tar 过来的。碰到这种情况该怎么办呢？在上面我们讲过，每个 ORACLE_HOME 下面都有本地 Inventory，在本地的 Inventory 中也已经注册了所有的 Oracle 组件的信息，那么我们就可以通过本地的 Inventory 来创建全局的 Inventory。

重建全局 Inventory 的方法很简单，第一步我们首先要编辑一个 oraInst.loc 文件，使之指向我们要创建全局 Inventory 的目录。

```
Inventory_loc=/opt/oracle/oraInventory
inst_group=oinstall
```

然后将目录转向 ORACLE_HOME 下的 oui/bin 目录：

```
$ cd $ORACLE_HOME/oui/bin
```

在该目录下执行下面的脚本就可以完成全局 Inventory 的创建：

```
%./runInstaller -silent -ignoreSysPrereqs -attachHome
ORACLE_HOME="<Oracle_Home_Location>" ORACLE_HOME_NAME="<Name_Of _Oracle_Home>"
```

下面是老白重新创建自己测试机上的 Inventory 的命令：

```
oracle@ bin]$ ./runInstaller -silent -ignoreSysPrereqs -attachHome
ORACLE_HOME="/opt/oracle/product/10g" ORACLE_HOME_NAME="ora10g"
Starting Oracle Universal Installer...

No pre-requisite checks found in oraparam.ini, no system pre-requisite checks will be
executed.

>>> Ignoring required pre-requisite failures. Continuing...

The Inventory pointer is located at /etc/oraInst.loc
The Inventory is located at /opt/oracle/oraInventory
'AttachHome' was successful.
```

这个命令支持 Oracle 10.2，对于 Oracle 9*i* 和 10.1 的版本，如果丢失了全局 Inventory，那么就需要从一个类似的平台克隆一个 Inventory 过来。克隆的具体方法由于篇幅有限，在这里就不做过多的描述了，有兴趣的朋友可以参考 Metalink 上的下面几个文档：

```
Note 559299.1 "Cloning An Existing Oracle9i Release 2 (9.2.0.x) RDBMS Installation
Using OUI"
Note 559301.1 "Cloning An Existing Oracle10g Release 1 (10.1.0.x) RDBMS Installation
Using OUI"
Note 559304.1 "Cloning An Existing Oracle10g Release 2 (10.2.0.x) RDBMS Installation
Using OUI"
Note 559305.1 "Cloning An Existing Oracle11g Release 1 (11.1.0.x) RDBMS Installation
Using OUI"
```

针对 11.2 以后的数据库，这个命令有所区别：

```
% ./runInstaller -silent -ignoreSysPrereqs -attachHome
ORACLE_HOME="<Oracle_Home_Location>"
```

我们不需要指定 ORACLE_HOME 的名字了。

刚才的例子是单节点的，在 RAC 环境下要稍微复杂一些。在本小节的最后一部分，我们来简单探讨一下重建 RAC 环境下的全局 Inventory 的方法。

10*g* RAC 引入了 CRS，因此我们在重建 Inventory 的时候，至少需要修复两个 ORACLE_HOME，一个是 RDBMS 的，一个是 CRS 的。命令如下：

```
./runInstaller -silent -ignoreSysPrereqs -attachHome ORACLE_HOME="<10g Ora_Crs_Home
Path>" ORACLE_HOME_NAME="<Name of oracleCRSHome>" LOCAL_NODE='node1'
CLUSTER_NODES=node1,node2 CRS=true
./runInstaller -silent -ignoreSysPrereqs -attachHome ORACLE_HOME="<10g Oracle_Home
Path>" ORACLE_HOME_NAME="<Name of oracleHome>" LOCAL_NODE='node1'
CLUSTER_NODES=node1,node2
```

1.2.2　口令文件

口令文件一般来说放在$ORACLE_HOME/dbs 目录下，在 Windows 平台下面，这个文件是在$ORACLE_HOME/database 目录下。

Oracle 数据库的口令文件存放有超级用户的口令及其他特权用户的用户名／口令。在创建一个数据库的时候，在$ORACLE_HOME/dbs 目录下会自动创建一个与之对应的口令文件。此文件是进行初始数据库管理工作的基础。在此之后，管理员也可以根据需要，使用工具 ORAPWD 手工创建口令文件。

默认安装下，最初的口令文件中只包含 sys 账号的信息，我们来看看老白的测试库的信息：

```
[oracle@ dbs]$ strings orapworcl
]\[Z
ORACLE Remote Password file
INTERNAL
87C5F4BF47942D0E
4CCF4A082AD3F312
```

这时候如果想把 system 账号设置为 sysdba 权限，我们来看看口令文件有什么变化：

```
[oracle@ dbs]$ sqlplus '/as sysdba'
SQL*Plus: Release 10.2.0.4.0 - Production on Fri May 6 12:18:25 2011
Copyright (c) 1982, 2007, Oracle.  All Rights Reserved.
Connected to:
```

```
Oracle Database 10g Enterprise Edition Release 10.2.0.4.0 - Production
With the Partitioning, Oracle Label Security, OLAP, Data Mining
and Real Application Testing options
SQL> grant sysdba to system;
Grant succeeded.
SQL> exit
Disconnected from Oracle Database 10g Enterprise Edition Release 10.2.0.4.0 -
Production
With the Partitioning, Oracle Label Security, OLAP, Data Mining
and Real Application Testing options
[oracle@ dbs]$ strings orapworcl
]\[Z
ORACLE Remote Password file
INTERNAL
87C5F4BF47942D0E
4CCF4A082AD3F312
SYSTEM
6971FF1AEA6387CE
```

这个文件中包含了 system 这个账号。同样我们也可以给其他账号分配 sysdba 权限。比如我们授予 scott 账号 sysdba 权限，授权后，在口令文件中就保留了 scott 账号的密码信息。这样 scott 账号就可以在数据库没有启动时进行鉴权了。

我们可以使用 orapwd 工具重建口令文件，重建的方法十分简单，我们就不在本节中多说了。

1.2.3　参数文件

参数文件一般来说放在$ORACLE_HOME/dbs 目录下，在 Windows 平台下面，这个文件是在%ORACLE_HOME%/database 目录下。早期的 Oracle 数据库的参数文件称为 PFILE，从 Oracle 9*i* 开始，引入了服务器参数文件 spfile。和 pfile 不同，spfile 采用了一种二进制的方式，同时保留了对原有的文本参数文件的支持。在$ORACLE_HOME/dbs 下存放 spfile 文件（如果是 Windows 平台，则是在%ORACLE_HOME%/database 下）。Oracle 启动的时候会按照如下顺序查找参数文件：

- $ORACLE_HOME/dbs/SPFILE.ORA
- $ORACLE_HOME/dbs/spfile.ora
- $ORACLE_HOME/dbs/init.ora

实际上，spfile 也不完全是二进制的，只是在原有的 pfile 基础上加入了一些二进制的管理和校验信息，直接编辑 spfile，可以取出其中的参数配置的所有信息。

在数据库启动的时候，数据库会根据搜索路径自动查找参数文件。如果找不到参数文件，数据库启动会失败。如果需要，可以使用下面的方法指定一个启动参数文件：

```
sqlplus /nolog
sql>connect sys/... as sysdba;
sql>startup pfile=$ORACLE_HOME/dbs/init.ora;
```

采用服务器参数文件后，参数文件的修改就相对容易了一些。可以通过 Oracle 提供的指令修改:

```
ALTER SYSTEM SET <PARAMETER>=<VALUE> SCOPE='SPFILE';
```

对于习惯于修改文件的人，可以使用下面方法实现类似 *8i* 的参数修改。首先通过下列语句创建一个文本的参数文件。

```
sql>create pfile='...' from spfile;
```

生成参数文件后可以手工修改。但是每次修改后要使用下面语句生成新的 spfile。

```
SQL>CREATE SPFILE='...' FROM PFILE='...';
```

1.2.4　控制文件

控制文件是 Oracle 数据库中十分重要的文件，Oracle 数据库启动时，首先会去读参数文件，读了参数文件，实例所需要的共享内存区和后台进程就可以启动了，这就是数据库实例启动的 nomount 阶段。完成这个步骤以后，就需要通过参数文件中的 control_files 参数，找到数据库的控制文件，然后打开控制文件，对控制文件进行校验。这就是 Oracle 数据库实例启动过程中的 Mount 阶段。

控制文件中包含了 Oracle 数据库中十分重要的信息，其中包括整个数据库的物理结构、所有数据文件、REDO LOG 文件等的信息。当然控制文件中还包含了一些其他的重要信息，比如归档模式下的日志归档情况、rman 备份时的 catalog 信息等。要想了解控制文件中包含哪些内容，可以通过下面的语句进行查询：

```
Select type,record_size,records_total,records_used from
v$controlfile_record_section;

TYPE                 RECORD_SIZE RECORDS_TOTAL RECORDS_USED
-------------------- ----------- ------------- ------------
DATABASE                     192             1            1
CKPT PROGRESS               2036             4            0
REDO THREAD                  104             1            1
REDO LOG                      72             5            3
DATAFILE                     180           100            8
FILENAME                     524           116           12
TABLESPACE                    68           100            9
TEMPORARY FILENAME            56           100            1
RMAN CONFIGURATION          1108            50            0
LOG HISTORY                   36           113           53
OFFLINE RANGE                 56           145            0
ARCHIVED LOG                 584            27           27
BACKUP SET                    40           101            0
BACKUP PIECE                 736           204            0
BACKUP DATAFILE              116           210            0
BACKUP REDOLOG                76            53            0
DATAFILE COPY                660           203            0
BACKUP CORRUPTION             44           185            0
COPY CORRUPTION               40           101            0
DELETED OBJECT                20           203            0
PROXY COPY                   852           210            0
BACKUP SPFILE                 36           113            0
DATABASE INCARNATION          56            72            1
```

我们可以看到，控制文件中包含了数据库信息、CKPT 进程信息、REDO 信息、数据文件和表空间信息等重要的数据库信息，也包含了日志切换的历史信息和 RMAN 备份的 CATALOG 信息。

1.2.5　在线日志文件

在线日志文件即 REDO LOG 文件，“在线”这两个字是用于和归档日志区分的。在线日志是数据库中十分重要的文件，主要用于记录数据库的变更信息。Oracle 使用在线日志文件记录数据库变更信息的目的是，当数据库实例宕掉的时候，可以通过在线日志文件中记录的信息进行恢复。

在线日志文件的存在，解决了数据库实例突然宕掉或者服务器宕机后的系统恢复问题。有了在线日志文件，就不用害怕 Oracle 数据库突然宕掉后数据库实例无法自动修复了，因为它的固有机制可以确保数据库完整恢复。

1.2.6　数据文件

数据文件是存储 Oracle 数据库中的数据的，也是 Oracle 数据库中最为核心的文件。Oracle 数据库中的表、索引等都是记录在数据文件中的。其中系统表空间包含的数据文件里保存了数据库的元数据（metadata），这部分数据是十分关键的，如果 metadata 出现故障，那么我们在访问数据库的数据时就会发生问题。

数据文件中还有一类特殊的文件，即临时文件，一般来说，临时文件属于临时表空间。临时文件是 Oracle 存放临时性数据的，比如排序数据、临时表。一旦数据库重启，临时文件中的内容将会丢失。因此，我们不能把永久性的表和索引存放在临时文件中。

1.2.7　归档日志文件

归档日志文件是用于长期保存的，它是在线日志的离线拷贝版本，当在线日志切换的时候，ARCH 进程就会将这个刚刚关闭的在线日志文件的内容复制到磁盘上，长期保存。归档日志的主要用途是用于数据库的恢复操作。进行数据库完全恢复或者不完全恢复的时候，需要将备份的数据文件恢复到硬盘上，然后通过归档日志将其前滚到所需要的时间点。

在设置了逻辑复制的环境中，归档日志也有可能用来进行挖掘，从而生成 LCR（逻辑变化记录），因此在配置了 STREAMS、GOLDENGATE 等逻辑复制的环境中，归档日志需要在磁盘上存储更长的时间，以便于逻辑复制使用。在这种环境中，保留 5 日以上的归档日志是十分必要的，如果你的存储空间足够大，请给予归档日志更大的存储空间，并且这些归档日志的删除策略也要做适当的调整，不能由备份软件自动删除，而是要通过一个定时任务，删除几天前的归档日志。

1.3　实例和多实例数据库

如果有人问“多实例数据库和 RAC 是什么关系”？我想能够正确回答出来的人不会太多。要回答好这个问题，首先需要知道什么是 Oracle 实例，Oracle 实例和 Oracle 数据库之间存在什么关系。大多数 DBA 都学习过实例的概念，实例是访问 Oracle 数据库的通道，包含共享内存和后台进程。一个 Oracle 实例一次只能打开一个 Oracle 数据库，而一个 Oracle 数据库可以同时被多个实例打开。被多实例打开的 Oracle 数据库，必须是一个 RAC 数据库。

更进一步讲，RAC 应该是 Oracle RDBMS 的一个选件，但其实 RAC 数据库的说法并不严谨，不过我们大可不必过分纠结于此问题。本节将较为全面地介绍多实例数据库。

1.3.1　什么是数据库实例

在实际的开发应用中，关于 Oracle 数据库，经常听见有人说建立一个数据库、建立一个实例（Instance）、启动一个实例之类的话。其实问他们什么是数据库、什么是实例，很可能他们会说“数据库就是实例，实例就是数据库啊，没有什么区别”。我只能说虽然他们使用了很长时间的 Oracle 数据库，也算积累了一定的经验，不过基础的概念还是不太清楚。

我们都知道，数据库就是存储数据的一种媒介，数据都是按照某种格式存储在某些特定的文件中的。我们比较早接触过的数据库，比如说 DBASE、FOXBASE 之类的，实际上更多地表现为一种记录管理系统，在文件中按照既定的格式存储了一些数据记录。早期开发 DBASE 应用的程序员可能还记得，最初的 DBASE 数据库都是单机环境的，因此在开发时不用考虑互斥和锁的问题，只需要按照业务要求处理记录就行了。后来有了 NOVELL 网，FOXBASE 得到了广泛的应用。老白最早学习 NOVELL 网的时候也很困惑，对于 FOXBASE 能够在 NOVELL 网上运行感到十分神秘，后来明白了实际上 NOVELL 网为 FOXBASE 数据库提供了一个共享文件系统，FOXBASE 正是通过 NOVELL 的共享文件系统在多个网络终端之间同步，这样一来在 NOVELL 网上的 FOXBASE 应用就十分容易理解了。在 FOXBASE 里，我们通过 LOCK TABLE 命令来锁定某张表，然后再对其进行访问和修改，这样就可以避免多用户环境下的冲突问题了。

刚才似乎扯得有点远了，反过头来，我们再来认识 Oracle 数据库。Oracle 数据库是一种支持高并发的 RDBMS 系统，因此 Oracle 也需要解决在大量并发用户下的一致性访问问题。Oracle 数据库对外提供的访问方式并不是应用程序直接打开数据文件来操作数据库，而是通过一种 TWO-TASK 的模式提供服务。在这种架构下，应用无法直接访问数据库，而必须通过一种被称为实例（Instance）的逻辑结构去访问数据库。

于是我们需要了解一个十分重要的新概念——实例。那么什么是实例呢？官方的说法，实例指的就是操作系统中一系列的进程以及为这些进程所分配的内存块。如果用更容易理解的方式来解释，那就是说 Oracle 数据库的实例是我们访问 Oracle 数据库的通道。

下面我们先来了解一些数据库实例的基础概念。在一个数据库实例中，包含了 Oracle 的共享内存和一系列的后台进程，一个实例在同时只能打开一个数据库，而一个数据库可以同时被多个

实例打开，当然了，这种情况就是我们常说的 RAC。Oracle 数据库的实例必须依赖于某个特定的 ORACLE_HOME，启动实例需要的所有的程序和相关的文件（除了数据库外）都包含在 $ORACLE_HOME 中。除此之外，每个实例都有自己独立的 SID。在同一个 ORACLE_HOME 下，允许启动多个实例，但是这些实例必须拥有不同的 SID。另外，一个非 RAC 的数据库是不允许被多个实例打开的，因为实例 mount 独立数据库的时候，是以排他的方式进行的。一个 RAC 的数据库，只允许同一个 RAC 集群中的多个实例打开，非相同集群的节点是无法打开同一个数据库的。

通过上面的介绍，我们了解了实例、数据库、ORACLE_HOME、SID 这些概念之间的关系，下面我们进一步来了解实例。在 Oracle 中，我们可以启动一个 Oracle 的实例，这个时候虽然有了进程还有 SGA 等一系列的内存块，但是并没有把数据库文件读取进来。实例启动后，为访问 Oracle 数据库的应用提供了一个基础的环境。这个基础环境包含了一组共享内存，后者又包含了 Oracle 数据库的一系列内部数据结构，也包括了 Oracle 数据库的 SQL 和数据字典缓冲（共享池中的库缓存和字典缓存），还有数据块缓冲（DB Cache）。在一个启动了 Oracle 实例的 UNIX 系统上，使用 ipcs 命令可以看到共享内存的情况：

```
[oracle@ ~]$ ipcs
------ Shared Memory Segments --------
key          shmid      owner      perms      bytes         nattch status
0x9a8837b4   163840     oracle     640        1050673152    20
------ Semaphore Arrays --------
key          semid      owner      perms      nsems
0x03c13d24   753664     oracle     640        154
```

我们可以看到，Oracle 用户分配了一组共享内存和信号灯，这和我们在 UNIX 下编写具有共享内存的应用程序并无不同。实例启动时首先会装载参数文件，根据参数文件中定义的内存相关参数创建共享内存和信号灯，然后将参数文件装载到共享内存中被称为 ksppi 的内存区域，同时将当前实例的参数装载到一个独立的区域——ksppsv。根据参数文件，启动进程完成 SGA 中内存结构的初始化工作，然后启动相关的后台进程。这个过程完成后，数据库实例启动的第一步 nomount 就完成了。

实例启动到 nomount 后，所有的共享内存和后台进程就都已经装载完毕。于是，系统根据参数文件中控制文件的位置，打开控制文件，并对控制文件进行校验，如果这个步骤没有发现问题，就完成了 mount 步骤。

数据库实例 mount 完成后，通过对控制文件、UNDO 和 REDO 等进行比对分析后，发现数据库状态是一致的，数据库实例就可以打开数据库了。数据库打开后，数据库实例就可以提供对外服务了。

数据库实例启动过程在很多书中都有十分详细的介绍，因此本节就不再赘述。数据库实例启动后，应用程序就可以通过数据库实例来访问数据库了。

应用要访问 Oracle 数据库，可以通过三种方式：第一种方式是应用进程直接访问数据库实例的共享内存，第二种方式是通过 beq 协议在本机上访问，第三种方式是通过网络协议访问。第一

种方式使用的场合很少，我们不做讨论。下面着重讨论通过第二种和第三种方式访问数据库。

首先，后两种访问数据库的方式都是基于 two-task 结构的，都需要在数据库服务器上建立一个服务进程（server 进程，或者前台进程）来为客户端应用服务（在这里我们只讨论独立服务器模式，共享服务器模式十分类似，我们将在后面进行专题描述）。two-task 架构下访问数据库，首先需要在服务器端创建一个进程，这个进程启动时先要映射共享内存，然后才能够通过共享内存中的内部数据结构完成会话的初始化工作。

在本机上不经过 SQL*Net 连接数据库，前台进程和用户进程之间通过 IPC 机制进行通信，通信协议就是著名的 Bequeath 协议，简称 BEQ 协议。而如果通过 SQL*Net 连接数据库，那么就需要使用网络协议。现在 TCP/IP 协议已经成为使用最为广泛的协议，因此我们主要面对的是 TCP/IP 协议，而 10 多年前，著名的 SPX 协议、DECnet 协议、Token Ring 协议等都曾经是 DBA 进程配置的协议。使用 SQL*Net 协议的前台进程和用户进程之间的通信采用 Socket 通信。实际上，在服务器上，我们也可以使用 SQL*Net 连接数据库，只不过我们很少会去这样做，因为 BEQ 协议在效率上高于 Socket 通信。

除了使用的协议不同，在本机上通过 BEQ 协议连接数据库实例和通过 SQL*Net 连接数据库实例还有什么不同吗？很多 DBA 可能会感觉有所不同，因为在本机上直接连接数据库协议，前台进程是 shell 进程产生的子进程；而通过 SQL*Net 连接数据库实例，server 进程是 LISTENER（tnslsnr）产生的子进程，如果监听器进程的属性不同，那么产生的子进程会和 shell 直接产生的子进程有所不同。这一点不同在早期的 Oracle 版本中确实存在，而自从$ORACLE_HOME/bin/oracle 这个映像文件被设定为 s 属性后，这个问题就不存在了。Oracle 映像通过 s 属性可以将子进程的属性转为 Oracle 用户。

不过使用 BEQ 协议和网络协议在服务进程方面还是有所不同的，BEQ 协议通过 IPC 通信，因此不需要使用 Socket，而通过网络协议（SQL*Net）连接，客户进程最初连接的是 tnslsnr 进程，tnslsnr 进程接受了连接请求后，为其创建一个子进程，然后通知客户端进程重新连接到子进程上继续工作。在这个时候，就存在一个 Socket 重定向的问题。监听器产生子进程时会为新的连接分配一个未被使用的端口号，这个子进程启动后就在该端口上侦听，同时监听器会通知客户端进程，要求其重定向到新的端口号。此时客户端进程会关闭老的 Socket，并打开一个新的 Socket，完成登录操作。

可能有些 DBA 对上面的讨论感到有些迷茫了，怎么讨论实例的问题，一下子又转到了监听器的工作机制上了，这些知识对于 DBA 又有什么作用呢？我们刚才一直在强调实例是客户端访问数据库的通道，因此讨论客户端如何通过监听器连接数据库就十分有意义了。

首先第一点，现在很多客户对系统安全性要求很高，因此服务器上大量的网络端口是被封掉的，只有必须使用的才会被开放。那么对于 Oracle 数据库来说，我们只需要开放监听器所需要的端口就可以了吗？事实上不是这样的，除了开放类似 1521 的监听端口外，我们还需要开放一些高端口，这些端口将被用于 Socket 重定向。

可能很多用户在客户端连接数据库的过程中经常会碰到 TNS-12535 之类的错误，开始的时候总是从网络超时的角度去分析，不过这样分析往往很难找到真正的故障原因。这类问题在一个

存在防火墙的环境中，往往是由于在防火墙环境下的 Socket 重定向引起的，特别是在有 NAT 功能的防火墙上，这类问题很容易出现。客户端连接的是一个 NAT 翻译后的 IP 地址，而重定向的时候，监听器要求客户端连接到真实的 IP 地址上，这样就会出现连接超时导致的失败。这种情况一般的解决方案是使用 connect manager（CMAN），老白也曾经碰到过几个这样的客户，最后都是通过 CMAN 来彻底解决问题的。

1.3.2　多实例数据库

多实例数据库的概念同样经常让 DBA 感到迷惑。多实例数据库可以说是 RAC 的另外一种称呼，Oracle RAC 的特点就是多个数据库实例可以同时打开相同的数据库，进行并发的操作。多实例数据库是 Oracle 高可用架构和高可扩展性架构的核心技术，多个实例同时打开数据库进行读写，可以避免某个实例故障导致的系统不可用，同时多实例实现负载分担，也可以减轻某个实例的工作负载，从而提高整体吞吐能力。

Oracle RAC 是多实例数据库的正式名称，RAC 具有很多激动人心的特性，但是实施 RAC 也有一些潜在的风险。在实施之前，我们必须认真地了解 RAC 的一些基本概念。

RAC 多实例数据库架构有几个基础。第一个基础是共享存储，多台服务器（一般来说我们称之为节点）可以同时并发读写相同的文件。如果没有这个基础，RAC 就无从谈起。我们可以使用很多技术来实现共享存储，最为普遍使用的就是存储局域网络 SAN，通过光纤交换机连接的共享存储，一组 lun 可以同时被多个服务器节点访问。除了 SAN 外，以 NETAPP 为代表的基于 TCP/IP 的存储解决方案也是 RAC 的可选方案，通过 ISCSI 或者 NFS 共享文件系统，同样可以满足 RAC 对于共享存储的需要。在底层的可共享硬件的基础上，对于裸设备或者除了 NFS 外的非共享文件系统，RAC 还需要依赖于操作系统提供的并发存储支持，允许软件并发访问底层存储。比如说 IBM 的 HACMP 提供的并发 VG 的支持，允许同一个 VG 在多个节点上同时被激活。另外在此基础上的集群文件系统（CFS）也是实施部署 RAC 的不错的平台。从 10*g* 开始，Oracle 也提供了一个自己的共享存储解决方案 ASM，在底层提供共享的存储硬件的基础上，Oracle 可以不使用第三方的共享存储解决方案，仅仅使用自己的 ASM 技术，实现类似 HACMP 并发 VG 或者赛门铁克 CFS 的功能。

除了共享存储外，RAC 还需要依赖于 CLUSTERWARE。在 10*g* 之前，RAC 必须使用各个厂商提供的 CLUSTERWARE，比如 IBM 的 HACMP、HP 的 MC/SG 和 TRU64 CLUSTER。另外也可以使用第三方厂商提供的 CLUSTERWARE，比如赛门铁克的 STORAGE FOUNDATION 中的 CLUSTER 组件。从 Oracle 10*g* 开始，RAC 完全可以脱离第三方的 CLUSTERWARE，而使用 ORACLE 自己的 CRS。CRS 是 Cluster Ready Services 的简称，CRS 提供了 RAC 数据库系统所必需的运行环境。而且从 10*g* 开始，RAC 数据库（RAC RDBMS）必须依赖于 CRS，无论你是否安装了第三方的 CLUSTERWARE，RAC RDBMS 的底层堆栈只能在 CRS 的基础上运行。

这里有一个读者很容易混淆的概念，就是 CRS 和 RAC 之间的区别。很多 DBA 认为 CRS 就是 RAC，RAC 就是 CRS，实际上这是一个十分错误的认知。RAC 更严格地说是 RAC RDBMS，

RAC 只是 Oracle RDBMS 的一个选件，安装并启用了 RAC 功能的数据库我们称之为 RAC RDBMS，这是一个多实例的关系型数据库系统。而 CRS 只是一个 CLUSTER 的组件，它提供了 RAC RDBMS 运行所必需的底层集群环境。CRS 本身也不提供共享存储系统，它只提供了 CLUSTER 的节点管理、健康性检查以及一系列 CLUSTER 应用（比如 VIP、ONS 等应用），真正在 RAC 中提供共享存储的是 ASM、HACMP 或者 CFS 等技术。

一个多实例的数据库系统，必须运行在 CLUSTERWARE 环境中，同时 CLUSTERWARE 也对 RDBMS 的运行情况进行一定限度的监控。以 10*g* CRS 为例，RAC RDBMS 启动时必须依赖 CLUSTERWARE 环境，但是反过来，CRS 可以不知道 RAC RDBMS 的存在，也就是说一个 RAC 数据库实例可以不经过在 CRS 的 OCR 中注册而独立启动，而不会影响 RDBMS 的正常功能。但是如果 CRS 不知道这个数据库的存在，那么这个数据库及其实例就都不在其监控之中，一旦某个数据库实例出现故障，CRS 也不会做出相应的反应，比如重启节点等。

多实例数据库系统的一个很重要的特性是多个实例可以并发对同一个数据进行读写，这一点也是 RAC 十分核心的功能。不过在多实例数据库环境中，每个实例都拥有自己独立的 SGA，为了确保数据库的一致性，Oracle RAC 系统需要使用一个被称为缓冲区融合（CACHE FUSION）的技术来实现多个节点上的缓冲区的一致性访问。因此，在多实例数据库中修改数据，需要一些额外的成本。除了 Oracle 锁（ENQUEUE）被扩展为全局资源外，如果一个 BUFFER 在多个实例中被访问，那么这个 BUFFER 也就会成为全局 BUFFER。对于全局 BUFFER 的访问，其开销是要比普通的 BUFFER 大一些的，因为每次访问都需要向某个 BUFFER 的 MASTER 节点咨询该 BUFFER 的情况，并由 MASTER 节点来授权对该 BUFFER 的各种访问。在早期的 Oracle RDBMS 版本中，某个 BUFFER 的 MASTER 节点从该 BUFFER 被装载到被换出，是不会发生变化的。这样一种机制，导致了某个 BUFFER 的 MASTER 节点不一定是访问这个 BUFFER 最多的节点，从而也导致了一些不必要的网络包用来处理 MASTER 节点和访问节点之间的交互请求信息。从 10.0.1.2 开始，Oracle 提供了一种新的机制，即动态 REMASTER 机制 DRM，这个机制的出发点是某个资源的 MASTER 节点不是一成不变的，而是根据该资源被某个节点的访问频率的改变，自动进行动态的 REMASTER。这种技术的出发点是十分好的，在一个设计良好的系统中，或者在一个负载并不是很高的多实例系统中，DRM 能够发挥很好的作用。很多从 9*i* 升级到 10*g* 的系统，升级后都发现 RAC INTERCONNECT 的流量以及 GES/GCS 的相关指标都有所改善。不过对于一些比较繁忙、写操作很多的系统，DRM 技术可能导致很多的问题。比如说，节点启动时 RECOVERY 的性能大幅度下降，OPEN 数据库很慢，或者在一个错误节点上执行了一个大批量数据修改操作时，大量的 REMASTER 可能导致系统短暂 HANG 住 。

多实例数据库的特性对于 DBA 分析数据库故障，以及进行优化处理都提出了一个新的要求。就是我们在分析某个事件或者问题时，不能局限于某个实例，而要将思路拓宽到所有实例。比如说我们发现某个操作被 HANG 住时，不仅仅要分析本实例上面可能存在的 BLOCKER，而且要将分析的范围扩大到其他所有的实例，才有可能找到问题的根源。

为了实现多实例数据库技术，Oracle 的很多组件都带有实例特性。Oracle 的所有数据文件都是所有实例共享的，一般来说必须将数据文件存放在所有实例都可以访问的存储系统上。有些

DBA 就可能有疑问了，如果某个数据文件，只有某个实例才会去访问（比如这个数据文件相关的表空间上的表，被限制为只在某个实例上访问），那么这个文件也必须要在所有节点上共享吗？回答当然是肯定的。虽然说在正常情况下，只有某个节点才会访问这个数据文件，但是一旦这个实例宕了，其他节点要替这个节点做 RECOVERY 的时候，这个文件就必须被其他节点访问了。在维护 RAC 数据库的时候，DBA 经常会犯的一个错误就是在添加裸设备的时候，往往只注意了在要添加的节点上修改文件的 OWNER 属性，而忘记了在其他节点上修改属性。一旦其他节点读取了这个文件，发现这个文件无法访问，那么在其他节点的 SGA 中，这个文件就变成无法访问了。这个时候哪怕我们修改了文件属性，在 SGA 中的文件状态还是无法改变的。有的 DBA 想通过设置文件 OFFLINE/ONLINE 来解决这个问题，不过这个办法好像也是无效的。对于这个问题，Oracle 官方建议一般是重启其他的几个实例。但是在一个 24×7 的生产系统中，重启实例是灾难性的，这种情况下执行一下 ALTER SYSTEM CHECK DATAFILES;可以让实例重新校验所有 ONLINE 文件的状态，恢复文件的可用性。

多实例的数据库中，每个实例拥有一组独立的在线日志记录，也就是我们常说的 REDO THREAD。每个实例独立生成在线日志信息，并且拥有独立的 LGWR 进程用于写入在线日志文件。但是在 RAC 数据库环境中，在线日志文件也必须是所有节点都能够共同访问的。原因也是一样的，当进行实例恢复的时候，由于相关的数据被写在多个在线日志文件中，因此必须用到所有的 REDO LOG THREAD 中的在线日志文件，才能够完成恢复。当我们在数据库上增加一个新的实例的时候，必须为这个实例创建一组新的在线日志记录，同时激活这个 REDO LOG THREAD。反过来，要从数据库中删除一个实例的时候，我们必须关闭这个 THREAD，否则无论这个实例是否被使用，数据库恢复的时候，仍然会需要使用这个 THREAD 的日志。在这种情况下关闭某个 THREAD 后重新做一次全库备份，会少很多麻烦事。如果你真的碰到了这种情况，而那个实例的在线日志文件还没删除，那么可以找找数据库恢复所需要的 CHANGE#是否在在线日志中存在，如果存在，你也可以直接恢复这个在线日志来解决这个问题。

在使用 UNDO 自动管理的模式下，每个实例都需要使用独立的 UNDO 表空间，这些表空间的数据文件也必须存放在所有实例都能够访问的共享存储上，其原因我们在前面已经多次提到，不再重复了。

在多实例数据库环境中，临时表空间是可以多个实例共享的，不过能够共享的仅仅是临时表空间，临时段是不能共享的。在一个临时表空间上，每个实例必须拥有自己独立的临时段。当临时表空间满的时候，如果其他实例的临时段有空闲空间，那么这个实例可以从其他实例的临时段中偷取一个 EXTENT，用于扩展自己的临时段。

1.4 数据库后台进程

数据库的进程可以简单地分为前台进程和后台进程，前台进程是 Oracle 客户端访问数据库而创建的影子进程，后台进程是维持 Oracle 数据库正常运行所必需的。本节我们将探讨各类数据库进程之间是如何分工协作，并实现高效访问的。

1.4.1　进程结构

进程是操作系统中的一种机制，它可使用操作系统中的资源完成某个特定的任务。一个进程通常有其专用的存储区和特定的功能。Oracle 进程体系结构的设计目的是尽可能地使用系统的资源，使访问者获得最大的吞吐量和最短的响应时间。

Oracle 实例有两种类型：单进程实例和多进程实例。单进程 Oracle（又称单用户 Oracle）是一种数据库系统，一个进程执行全部 Oracle 代码。Oracle 数据库和用户应用程序无法实现进程分离，所以 Oracle 代码和用户数据库应用程序都是通过单进程执行。

在单进程环境下的 Oracle 实例，仅允许一个用户存取，例如在 MS-DOS 上运行 Oracle 。

多进程 Oracle 实例（又称多用户 Oracle）使用多个进程来执行 Oracle 的不同部分，对应于每一个连接的用户都有一个进程。

在多进程系统中，进程分为两类：用户进程（又称前台进程）和 Oracle 后台进程。当用户运行一个应用程序时，如 PRO*C 程序或 Oracle 工具（如 SQL*Plus），系统会为用户运行的应用建立一个用户进程，该进程通过某种方式启动一个服务器进程（前台进程），用于处理连接到该实例的用户进程的请求。如果应用和 Oracle 在同一台机器上运行，而不通过网络，那么用户进程和服务器进程之间可以通过 BEQ 协议通信，从而降低系统开销。然而，当应用和 Oracle 运行在不同的机器上时，用户进程使用 TCP/IP 协议，通过服务器进程访问 Oracle，可执行下列任务。

- 对应用所发出的 SQL 语句进行语法分析和执行。
- 从磁盘（数据文件）中读入必要的数据块到 SGA 的共享数据库缓冲区（该块不在缓冲区时）。
- 将结果返回给应用程序处理。

为了使系统性能最好并能够协调多个用户，多进程系统使用了一些附加进程，称为后台进程。在许多操作系统中，后台进程是在实例启动时自动建立的。一个 Oracle 实例可以包含许多后台进程，但它们不是一直存在的。后台进程的名称为：

- DBWR，数据库写入程序；
- LGWR，日志写入程序；
- CKPT，检查点；
- SMON，系统监控；
- PMON，进程监控；
- ARCH，归档；
- RECO，恢复；
- LCK*n*，封锁；
- D*nnn*，调度进程；
- S*nnn*，服务器。

每个后台进程与 Oracle 数据库的不同组件进行交互，完成特定的功能。比如，DBWR 负责脏数据存盘工作，LGWR 负责将 LOG BUFFER 中的数据写入在线日志文件。

1.4.2　后台进程的功能作介绍

Oracle 的后台进程负责管理和维护 Oracle 实例。每个后台进程都负责一项独立的工作。这些后台进程互相协作，完成 Oracle 的公共职能。它们之间会互相监控，一旦发现核心的后台进程出现异常，会主动关闭实例。下面依次介绍主要后台进程的功能。

1. DBWR 进程

DBWR 进程执行将数据块缓冲区写入数据文件的工作，是负责缓冲存储区管理的一个 Oracle 后台进程。在修改 DB Cache 中的某个缓冲区时，会将它标志为“DIRTY”，DBWR 的主要任务是将这些标为“DIRTY”的缓冲区写入磁盘，使缓冲区保持“CLEAN”。由于缓冲区填入数据库或被用户进程弄脏，未用的缓冲区数目会减少，最终可能导致用户进程从磁盘读入块到内存存储区时无法找到未用的缓冲区。DBWR 将管理缓冲存储区，使用户进程总能得到未用的缓冲区。

Oracle 采用 LRU 算法（最近最少使用算法）保持内存中的数据块是最近使用的，使 I/O 最小。下列情况预示 DBWR 要将弄脏的缓冲区写入磁盘。

- 当服务器进程将一缓冲区移入“DIRTY”链，此“DIRTY”链达到临界长度时，该服务器进程将通知 DBWR 进行写入操作。这个临界长度是数据库隐含参数_DB_BLOCK_WRITE_BATCH 值的一半。
- 服务器进程在 LRU 表中查找可用的数据块缓冲时，如果在查找了参数_DB_BLOCK_MAX_SCAN_CNT 所定义数量的缓冲区后，仍没有查到未用的缓冲区，那么它将会停止查找，并通知 DBWR 进行数据写入。DBWR 每次休眠时都会设置定时器，如果出现超时（每次 3 秒），DBWR 将通知自身。当出现检查点时，LGWR 将通知 DBWR 进行写入操作。在前两种情况下，DBWR 将“DIRTY”链中的块写入磁盘，每次可写的块数由初始化参数_DB_BLOCK_WRITE_BATCH 所指定。如果“DIRTY”链中没有该参数指定块数的缓冲区，DBWR 将从 LRU 表中查找另外一个“DIRTY”缓冲区。
- 如果 DBWR 在 3 秒内未活动，则出现超时。在这种情况下，DBWR 对 LRU 表查找指定数目的缓冲区，将所找到的任何弄脏的缓冲区写入磁盘。每当出现超时，DBWR 就查找一个新的缓冲区组。每次由 DBWR 查找的缓冲区的数目是隐含参数_DB_BLOCK_WRITE_BATCH 值的两倍。如果数据库空运转，最终 DBWR 会将全部缓冲区存储区写入磁盘。
- 在出现检查点时，LGWR 指定一修改缓冲区表必须写入到磁盘，而 DBWR 负责将指定的缓冲区写入磁盘。

在某些平台上，如果有多个 CPU，那么一个实例可设置多个 DBWR。在这样的实例中，DB Cache 被分为多个区，每个 DBWR 管理一个或者几个 DB Cache 分区。这种结构可以让一些数据块写入一个磁盘，另一些数据块写入其他磁盘，从而提升并发写入的性能。参数 DB_WRITERS 可以控制 DBWR 进程的个数。

2. LGWR 进程

LGWR 进程是负责管理日志缓冲区的一个 Oracle 后台进程，它将日志缓冲区写入磁盘上的日志文件。LGWR 进程将自上次写入磁盘以来的全部 REDO LOG ENTRY 写入到 REDO LOG 文

件中。触发 LGWR 写操作的条件如下。

- 当用户进程提交一事务时写入一个提交记录。
- 每 3 秒将日志缓冲区输出。
- 当日志缓冲区的已满 1/3 时，将日志缓冲区输出。
- 当 DBWR 将修改缓冲区写入磁盘时，则将日志缓冲区输出。
- 当 log buffer 达到 1MB 时。

LGWR 进程同步地写入到活动的镜象在线日志文件组。如果组中一个文件被删除或不可用，LGWR 可继续地写入该组的其他文件，而数据库实例可以继续运行。

日志缓冲区是一个循环缓冲区。当 LGWR 将日志缓冲区的日志项写入日志文件后，服务器进程即可将新的日志项写入到该日志缓冲区。LGWR 的写入速度很快，以确保日志缓冲区总有空间可写入新的日志项。

注意　当需要更多的日志缓冲区时，LWGR 会在一个事务提交前就将日志项写出，而这些日志项仅当后续事务提交后才永久化。

Oracle 使用快速提交机制，当用户发出 COMMIT 语句时，一条 COMMIT 记录立即被放入日志缓冲区，但相应的数据缓冲区改变被延迟，直到更有效时才将它们写入数据文件。提交时，将事务赋给一个系统变更号（SCN），它同事务日志项一起被记录在日志中。

3. CKPT 进程

CKPT 进程在检查点出现时，对全部数据文件的文件头进行修改，并在控制文件中记录该检查点。在早期版本中，该任务由 LGWR 执行。然而，在检查点明显地降低系统性能时，可使 CKPT 进程运行，将原来由 LGWR 进程执行的检查点的工作分离出来，由 CKPT 进程执行。从 Oracle 8 开始，CKPT 进程被独立出来，它不再将块写入磁盘，该工作转由 DBWR 完成。在 Oracle 7 中，可以通过初始化参数 CHECKPOINT_PROCESS 来控制是否启用 CKPT 进程协助 LGWR，它的默认值是 FALSE；另外，如果数据文件的数量很多，那么启用 CKPT 会对性能有一定的提升。

4. SMON 进程

SMON 进程负责在实例启动时执行实例恢复，并清理不再使用的临时段。在具有并行服务器选项的环境下，SMON 对有故障的 CPU 或实例进行实例恢复。从 Oracle 9*i* 开始，事务回滚操作的默认行为也是由 SMON 来负责处理的。虽然 SMON 本身不做恢复操作，而是启用并行进程来处理，但是它起到整体协调的作用。SMON 进程有规律地被唤醒，并检查是否有工作要完成，如有需要，就做相应的处理，否则继续休眠。

5. PMON 进程

PMON 进程在用户进程出现故障时执行进程恢复，负责清理存储区和释放该进程所使用的资源。比如，某个进程死掉了，PMON 要重置活动事务表的状态，释放锁资源，将该故障的进程 ID 从活动进程表中移去。PMON 还周期性地检查调度进程（DISPATCHER）和服务器进程的状态，如

果发现这些后台进程死掉，就需要重新启动。PMON 有规律地被呼醒，检查是否有需要完成的工作。

6. RECO 进程

RECO 进程是启用分布式选项时才会存在的进程，而且 DISTRIBUTED_TRANSACTIONS 参数大于 0。当然，分布式事务在绝大多数系统中是默认安装的，因此一般来说，总是能在数据库实例中看到这个进程。RECO 进程能够自动解决分布式事务中的故障。一个节点的 RECO 后台进程能够自动连接到包含错误的分布式事务的其他数据库中，在解决了所有的故障后，将这个全局事务从 dba_2pc_pending 等相关的表中删除。

当数据库服务器的 RECO 后台进程试图同一远程服务器建立通信时，如果远程服务器不可用，或者网络连接不能建立，RECO 将在一定时间间隔后自动重连。

7. ARCH 进程

ARCH 进程将已填满的在线日志文件复制到指定的存储设备。当数据库的日志模式为 ARCHIVELOG 模式并可自动归档时，ARCH 进程才存在。

8. LCK*n* 进程

LCK*n* 进程在具有并行服务器选件的环境下使用，可多至 10 个进程（LCK0，LCK1，…，LCK9），用于实例间的封锁。

9. D*nnn* 进程（调度进程）

D*nnn* 进程允许用户进程共享有限的服务器进程（SERVER PROCESS）。没有调度进程时，每个用户进程需要一个专用服务进程（DEDICATED SERVER PROCESS）。多线索服务器（MULTI-THREADED SERVER）可支持多个用户进程。如果系统具有大量用户，多线索服务器可以很好地支持，尤其在客户/服务器环境中。

在一个数据库实例中可建立多个调度进程。对每种网络协议至少建立一个调度进程。数据库管理员根据操作系统中每个进程可连接数目的限制决定启动的调度程序的最优数，在实例运行时可增加或删除调度进程。多线索服务器需要 SQL*Net 版本 2 或更高的版本。在多线索服务器的配置下，一个网络接收器进程等待客户应用连接请求，并将每一个发送到一个调度进程。如果不能将客户应用连接到一调度进程时，网络接收器进程将启动一个专用服务器进程，该网络接收器进程不是 Oracle 实例的组成部分，而是处理与 Oracle 有关的网络进程的组成部分。在实例启动时，该网络接收器被打开，为用户连接到 Oracle 建立一通信路径，然后每一个调度进程把连接请求的调度进程的地址传给它的接收器。当一个用户进程作连接请求时，网络接收器进程分析请求并决定该用户是否可使用一调度进程。如果是，该网络接收器进程返回该调度进程的地址，之后用户进程直接连接到该调度进程。有些用户进程不能调度进程通信（如果使用 SQL*Net 以前的版本），网络接收器进程不能将此类用户连接到一调度进程。在这种情况下，网络接收器将建立一个专用服务器进程和一种合适的连接。

1.4.3　哪些后台进程可以杀

在很多情况下我们需要杀死后台进程。比如，系统出现了大量挂起的现象，而通过

HANGANALYZE 工具分析，我们发现元凶是一个后台进程，那么是否要通过杀掉这个进程来解决问题，就要十分谨慎了。因为有些后台进程是不能随便杀的，一旦杀掉就可能导致数据库实例崩溃。因此，有些 DBA 给自己定了一条金科玉律，就是后台进程绝对是不能杀的。

其实这种做法过于保守了，只要你足够了解后台进程的主要功能，就可以十分安全地管理后台进程了。本节老白将以 Oracle 10*g* 和 11*g* 为例，和大家讨论究竟哪些后台进程是可以杀的。

首先我们来看六大核心进程。其实 Oracle 并没有核心进程这个概念，这是老白自己的归纳总结。那么哪几个后台进程可以称为六大核心进程呢？pmon、smon、dbwr、lgwr、reco 和 ckpt 这六个进程是所有 Oracle 数据库必不可少的，其中 ckpt 进程出现得较晚，其他五个进程是老白使用 Oracle 数据库以来就一直存在的系统进程。这些进程无论哪个出现故障，都会导致数据库实例崩溃。因此，这些进程是无论如何都不能杀的。如果我们杀掉其中某个进程，在 ALERT LOG 中就会发现各种错误。

在某个 shell 中，杀掉 ckpt 进程：

```
[oracle@localhost ~]$ ps -ef|grep ckpt
grid      5245     1  0 Sep25 ?        00:00:00 asm_ckpt_+ASM
oracle    5377     1  0 Sep25 ?        00:00:22 ora_ckpt_orcl
oracle   10891 10763  0 08:48 pts/4    00:00:00 grep ckpt
[oracle@localhost ~]$ kill -9 5377
```

执行了上述命令后，我们发现 ALERT LOG 中出现了：

```
Mon Sep 26 08:48:31 2011
System state dump requested by (Instance=1, osid=5342 (PMON)), summary=[abnormal
Instance termination].
System State dumped to trace file
/u01/app/oracle/diag/RDBMS/orcl/orcl/trace/orcl_diag_5365.trc
Mon Sep 26 08:48:32 2011
PMON (ospid: 5342): terminating the Instance due to error 469
Mon Sep 26 08:48:32 2011
ORA-1092 : opitsk aborting process
Dumping diagnostic data in directory=[cdmp_20110926084831], requested by (Instance=1,
osid=5342 (PMON)), summary=[abnormal Instance termination].
Instance terminated by PMON, pid = 5342
```

可以看到，由于 ckpt 出现故障，pmon 进程将实例关闭了。如果杀掉 pmon 又会出现什么情况呢？

```
Mon Sep 26 08:52:58 2011
Shutting down Instance (abort)
License high water mark = 4
USER (ospid: 11224): terminating the Instance
Instance terminated by USER, pid = 11224
Mon Sep 26 08:52:59 2011
Instance shutdown complete
```

我们看到，当 pmon 被杀掉后，一个前台进程执行了 shutdown abort 操作，终结了实例（Instance terminated by User）。可以看出，虽然 pmon 是监控进程的后台进程，但是一旦重要的后台进程出现故障，pmon 会自动关闭实例。反过来所有的 Oracle 进程，包括前台进程和后台进程，也在监

视 pmon，一旦发现 pmon 异常，会立即关闭实例。这种相互监控的机制也是大型系统中最为常用的方法。

下面我们来看看 10*g* 新增加的 MMAN 进程，MMAN（Memory Manager ）进程是 10*g* 新引入的进程，主要目的是实现共享内存自动管理的功能，自动调整共享内存各个组件的大小。

```
Mon Sep 26 11:09:52 2011
Errors in file /opt/oracle/admin/orcl/bdump/orcl_pmon_6261.trc:
ORA-00822: MMAN process terminated with error
Mon Sep 26 11:09:52 2011
PMON: terminating Instance due to error 822
Instance terminated by PMON, pid = 6261
```

可以看到，一旦 MMAN 进程出现故障，数据库实例就会崩溃。看样子 MMAN 是继六大核心进程之后的第七个不可杀的核心进程。

PSP0 进程在 10*g* 中开始引入，主要功能是启动其他的 Oracle 进程。这个进程也是一个十分关键的核心进程，一旦出现问题，将导致数据库实例故障。

```
Errors in file /opt/oracle/admin/orcl/bdump/orcl_pmon_32149.trc:
ORA-00490: PSP process terminated with error
Mon Sep 26 14:14:55 2011
PMON: terminating Instance due to error 490
Instance terminated by PMON, pid = 32149
```

至此，我们已经看到了八个关键核心进程。

下面我们来看一下 cjq0 进程。它是一个任务队列的调度进程，负责从 job$表中找到需要执行的任务，并分配 job 进程执行，如果 job 进程不足，会自动产生新的 job 进程（在 job_queue_processes 参数限制范围内）。下面来看看杀掉这个进程会有什么结果。

```
Mon Sep 26 09:07:18 2011
Restarting dead background process CJQ0
Mon Sep 26 09:07:18 2011
CJQ0 started with pid=25, OS id=12226
```

可以看出 cjq0 进程如果被杀掉，cjq0 进程会被重启。

既然 cjq0 都可以杀，那么 cjq0 产生的 JXXX 进程我们就不用做实验了，肯定是能杀的了。其实老白也是经常杀掉 JOB 进程的，在某些系统中，经常会有一些 JOB 进程占用大量的系统资源，从而导致数据库性能问题。这时，为了恢复 OLTP 应用的性能，杀掉 JOB 进程是最简单的方法。不过在杀掉 JOB 进程之前一定要做仔细的分析，如果 JOB 进程中正在做一个数据量很大的大型修改事务，那么杀掉这个 JOB，可能会导致大量的回滚操作，从而对系统性能产生更为不利的影响。

下一个我们要来研究的是 arch 进程，在 Oracle 10*g* 中，arch 进程一般是 arc0, arc1，...。我们来杀掉一个 arch 进程，看看会有什么结果。

```
Mon Sep 26 09:56:27 2011
ARCH: Detected ARCH process failure
ARCH: STARTING ARCH PROCESSES
ARC0: Archival started
```

```
ARCH: STARTING ARCH PROCESSES COMPLETE
ARC0 started with pid=16, OS id=6646
ARC0: Becoming the 'no FAL' ARCH
ARC0: Becoming the 'no SRL' ARCH
```

可以看出，arc0 进程被杀掉后，会自动重启，而数据库实例没有发生故障。因此 arch 进程如果出现故障，在必要情况下，我们是可以杀掉该进程的。

下面我们来看一下 CJQ0 进程，CJQ0 进程是队列监控同步进程（QMNC）和队列服务进程（Q*XXX*）的统称。

```
Mon Sep 26 09:10:46 2011
Restarting dead background process CJQ0
Mon Sep 26 09:10:46 2011
CJQ0 started with pid=25, OS id=12347
```

从上面的测试可以看出，CJQ0 进程是可以杀的，杀掉 CJQ0 进程的后果是相关进程重启。

MMON 和 M000 是 Oracle 10*g* 引入的新后台进程，MMON 是管理监控进程，M000 是 MMON 的 SLAVE 进程，协助 MMON 进程工作。如果这些进程出现故障会有什么结果呢?

```
Mon Sep 26 10:11:39 2011
Restarting dead background process MMON
MMON started with pid=11, OS id=11860
```

MMON 进程是可以自动重启的，当然也在可杀范围内了。

类似的 MMNL 进程也是 AWR 新增的进程，主要作用是将 AWR 数据从内存中刷新到表中。这个进程如果被杀掉也是可以自动重启的。在这里我们就不一一列出实验数据了：

- DISPATCHER 进程 D*XXX*：如果被杀掉，ALERT 会报错，不会导致实例宕机，根据需要进行重启。
- 共享服务进程 S*XXX*：如果被杀掉，不会导致实例宕机，根据需要进行重启。
- 并行进程 P*XXX*/PZ*XX*：并行进程，如果被杀掉，不会导致实例宕机，进程根据需要进行重启。
- 高级队列从属进程 Q*XXX*：如果被杀掉，不会导致实例宕机，进程根据需要进行重启。如果存在高级队列操作，杀掉此类进程要十分慎重。

Oracle 10*g* 引入了 ASM 后，也新增了一些和 ASM 有关的进程。首先 ASM 实例具有一系列的后台进程，其次，RDBMS 为了访问 ASM 也新增了一系列的后台进程。我们来看看 ASM 实例的后台进程。

ASM 实例也拥有类似 RDBMS 实例的核心进程，另外 ASM 实例新增了一些其他的后台进程，下面我们做一个简单的了解。

- ASMB：当数据库实例使用 SPFILE 时启动的 ASM 后台进程。这个进程是十分关键的，一旦出现故障将导致 ASM 实例宕机。
- RBAL：DISKGROUP 做 rebalance 的后台进程，该进程一旦有故障将导致 ASM 实例宕掉。
- DBW0：DB writes，和 RDBMS 的 DB WRITER 类似，不过是将 ASM CACHE 中的数据写入磁盘，该进程一旦有问题将导致 ASM 实例故障。

- SMON：恢复进程，类似于 RDBMS 的 SMON 进程，处理 DISKGROUP 的恢复操作，一旦有问题将导致 ASM 实例故障。
- CKPT：Checkpoint 进程，类似于 RDBMS 的 CKPT 进程，一旦有问题将导致 ASM 实例故障。
- PSP0：启动其他 ASM 实例进程的进程，一旦有问题将导致 ASM 实例故障。
- GMON：群组监控进程，用于节点监控和状态表的维护，一旦有问题将导致 ASM 实例故障。
- ora_ASMB：特殊的 ASM 前台进程。
- KATE：Konductor or ASM Temporary Errands，用来执行 ONLINE 磁盘的临时任务进程。
- VKTM：管理快速计时器的进程。
- PING：计量网络延时的进程。
- DIA?：类似于数据库的 diag 进程。
- DIAG：类似于数据库的 diag 进程。
- LGWR：Log writer, 和数据库类似，处理磁盘组的 REDO 信息。
- LMON：锁监控进程，类似于数据库的 LMON 进程。
- LMS?：锁监控 SLAVE 进程，类似于数据库的 LMS 进程。
- MMAN：SGA 自动调整进程，类似于数据库的 MMAN。
- b???：用于离线磁盘的 SLAVE 进程。
- x???：磁盘组重配置后删除磁盘的 SALVE 进程。
- pz??：用于 GLOBAL VIEW 查询的并行 SLAVE 进程。

Oracle 11*g* 在后台进程方面有了较多的改变，这种改变有时候甚至让我们感觉 Oracle 数据库变陌生了，需要重新认识 Oracle 11*g* 的后台进程结构。下面是新增的后台进程。

- ACMS（atomic controlfile to memory service），这是每个实例都有的代理进程，在 RAC 环境下，对控制进程事务的提交和回退起到辅助作用。
- DBRM（database resource manager），用于资源管理和资源计划相关的任务。
- DIA0（diagnosability process 0），目前只有“0”，没有其他进程，用于数据库的挂起检测和死锁处理。
- DIAG（diagnosability），进行诊断 DUMP，执行全局 oradebug 命令。
- EMNC（event monitor coordinator），用于数据库事件管理和发布的进程。
- FBDA（flashback data archiver process），闪回区归档进程，用于将跟踪表的历史记录写入归档日志，管理闪回数据区的空间。
- GTX0-*j*（global transaction），在 RAC 环境中为 XA 全局事务提供透明的服务，只在 RAC 环境中出现。系统会根据 XA 事务的负载情况确定启动几个这种进程。
- KATE，当磁盘离线时代理 ASM metadata 的 I/O。
- MARK，当写入一个离线磁盘出错时记录这个 ALLOCATION UNIT 为过期状态。
- SMCO（space management coordinator），执行空间管理有关的作业，比如空间预分配和空

间回收。可以动态生成 Wnnn SLAVE 进程。

- VKTM（virtual keeper of time），每秒更新一次时间，在高优先级情况下可以提供 20 毫秒的基准时间计数。
- DSKM（slave diskmon），用于 RDBMS 和 ASM 实例之间的联系通道。Master Diskmon 守护进程处理 I/O 隔离信息，I/O 资源管理计划将 Transaction Commit Cache 信息传输到 SAGE 存储，也用来在节点和 SAGE 存储服务器之间实现 skgxp ANT 协议。如果没有配置 SAGE 存储，diskmon slave 进程会在实例启动后自动关闭。

1.4.4 是谁在执行 SQL

“是谁在执行 SQL？”这个问题看似很简单，不过要认真考虑起来，却也不那么简单。很多工作了 7、8 年的 DBA 可能还真的没有认真考虑过这个问题。最初，老白一直以为是 Oracle 的后台进程在执行 SQL，然后将执行的结果返回给客户端进程。直到学习了 TWO TASK 这个概念后，才知道在客户端连接到数据库时，Oracle 会创建一个服务进程（Server Process），Oracle 的客户端通过和该进程通信来完成 SQL 的执行。

现在回头一想，如果是 Oracle 的后台进程来执行 SQL，那么在一个大型的数据库系统中，会有数千甚至上万个客户端在访问数据库，光凭几个 Oracle 后台进程是肯定无法完成这个任务的，这样就会出现瓶颈。从这个角度来看，服务进程应该是执行 SQL 的“最佳人选”了。在独立服务器模式下，每个 Oracle 会话都拥有一个独立的服务进程，如果让它来担当执行 SQL 的角色，那么就不会出现资源瓶颈了。

事实上，Oracle 也是这样安排的，TWO TASK 的服务器端的服务进程，担当的就是这样一个角色。客户端要执行的 SQL，通过 SQL*Net 或者 BEQ（客户端和数据库服务器运行在同一台服务器上时，可以不通过 SQL*Net，而直接通过 IPC 通信协议 BEQ 来通信）发送给服务进程，由它来完成 SQL 的执行。

我们可以更加深入地理解一下这方面的概念。先要问大家一个问题，执行 SQL 的主体是什么？可能很多 DBA 会觉得有些迷惑了，怎么什么简单的问题到了老白这里都变得那么不确定了呢？执行 SQL 的主体是会话（Session）呀，执行 SQL 的呼叫是由会话发起的，会话是 Oracle 数据库用户进行 SQL 操作的唯一渠道。

那么下一个问题又来了，会话又是什么呢？搞过网络编程的朋友可能早就听说过会话这个概念了。两个网络设备要进行通信，必须先建立起一个会话，这个会话就是承载所有通信工作的逻辑载体。再往前追溯，会话的概念来自通信行业，不过 Oracle 会话的本质更类似于网络设备之间的通信。大家都知道网络通信有面向连接的协议，也有面向无连接的协议。面向连接的协议一般适用于上下文和状态十分关键的应用场合。Oracle 的会话正是具有这样的特征，所以采用面向连接的通信协议，目前最常用的就是 TCP/IP 协议。出于安全考虑，在建立会话之初，Oracle 需要通过安全认证，这也就是我们常见的“数据库登录”，Oracle 的术语称为 LOGON。在 LOGON 的时候，首先 Oracle 客户端通过 SQL*Net 协议或者 BEQ 协议创建一个服务进程。如果通过 SQL*Net

协议，客户端首先要和监听器通过 TCP/IP、SPX 等面向连接的网络协议建立通信会话，由监听器创建一个服务进程，客户端进程将网络通信会话重定向到这个服务进程，随后，客户端进程和服务进程建立通信会话。大家可能注意到了，老白在这里说了很多次“通信会话”，这么说就是为了区别与 Oracle 的会话。和客户端通信的服务进程也就是 Oracle 术语中所说的前台进程，客户端和前台进程建立了通信会话并不等于说 Oracle 的会话已经建立了，会话是一个更为虚拟的概念。大家先不要着急，等老白一步一步地进行剖析。客户端和服务进程完成通信会话的握手后，首先将 LOGON 所需要的信息发送给前台进程。前台进程收到这些信息后，执行一个被称为 LOGON 的操作，校验用户和权限。这个校验工作可以有多种方式，最常见的是通过 SYSTEM 表空间中的 USER$表中保存的用户名和密码进行校验。Oracle 还支持其他方式的用户名、密码校验方式，比如操作系统校验、外部安全设备校验（LDAP）。完成 LOGON 操作后，系统会在数据库服务器的 SGA 中创建一个会话的数据结构，这个数据结构被称为 SESSION STATE OBJECT，Oracle 内部数据结构为 ksuse。在《Oracle RAC 日记》中，老白曾介绍过这个结构，具体如图 1-1 所示。

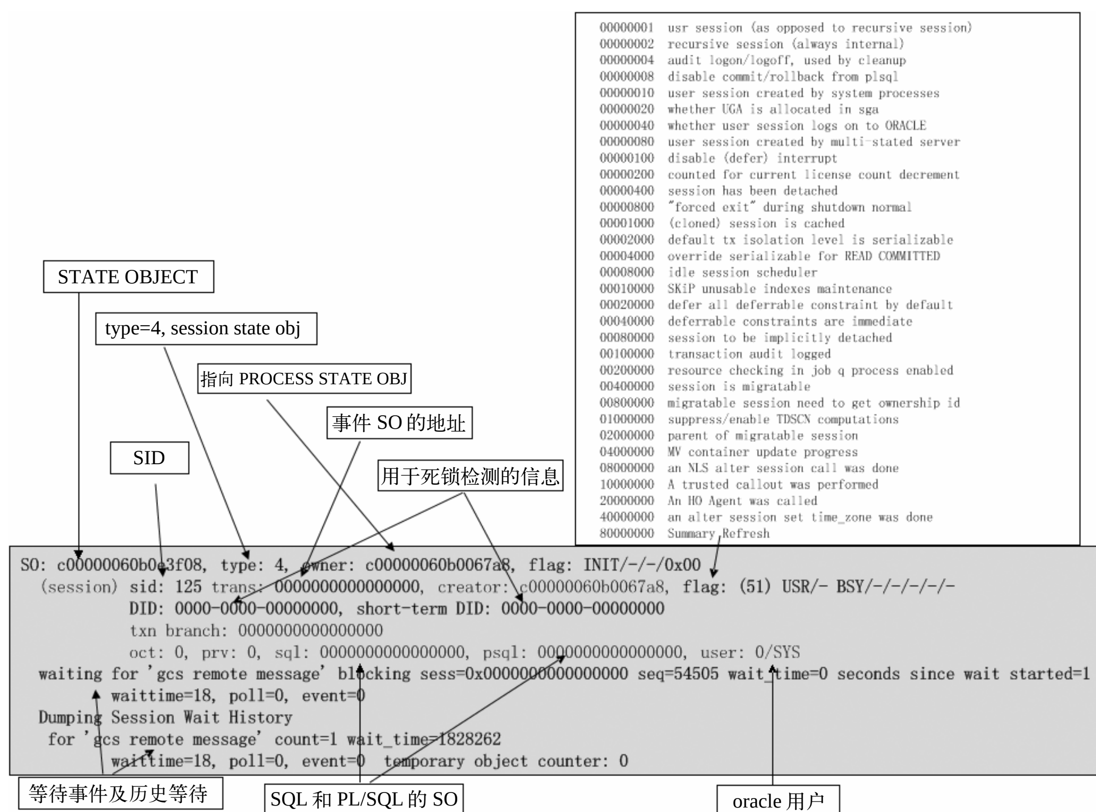


图　1-1

图 1-1 显示的是 SESSION STATE OBJECT 的信息，这个信息只包含了会话的基础信息，会话相关的数据结构十分复杂，已经超出了本书的讨论范围，因此不做更多的描述。

会话建立后，就成为客户进程和数据库实例之间的沟通渠道和桥梁，执行客户端对数据库的操作，包括执行 SQL。不过会话是一个逻辑结构，必须依赖于其容器——服务进程（也叫前台进程）。在独立服务器模式下，每个会话对应一个独立的服务进程，Oracle 也提供了一种共享服务器模式，在这种模式下，一个服务进程可以为多个会话服务，换句话说，一个服务进程可以成为多个会话的容器。

在共享服务器模式下，一般会配有一个或者多个调度进程（DISPATCHER，比如 D000）和一组服务进程，这些服务进程的最大数量受到 max_shared_servers 参数控制。当监听进程接收到来自客户端的连接请求时，不是创建独立的服务进程，而是使用现有的服务进程池来提供服务。监听进程首先会和调度进程通信，找到可用的服务进程，调度进程会将这个服务进程及其通信端口号发送给监听进程，监听进程将网络通信连接重定向到这个服务进程，完成客户端和服务进程之间的握手。

在共享服务器模式下，由于多个会话共享一个服务进程，因此服务进程作为共享资源，也可能成为一个瓶颈。如果使用同一个服务进程的几个共享会话中有一个执行了时间很长、开销很大的 SQL，那么其他会话将会处于等待状态。共享服务器模式在早期的数据库系统中的使用是比较广泛的，因为早期的服务器内存资源有限，如果使用独立服务器模式，内存资源将会十分紧张，而使用共享服务器模式可以在内存资源有限的情况下，支持大量的数据库会话。在目前内存资源十分充裕的情况下，共享服务器模式的应用就越来越少了，一个进程为一个会话服务，可以有效地提高系统总体的吞吐能力。

不过，共享服务器模式也并不是完全没有了用武之地，尤其是现在服务器性能有了大幅提高，对于某些没有使用连接池的，以短连接为主的应用来说，如果执行的大多数是开销较小的 SQL，那么使用共享服务器模式可以避免由于应用软件使用短连接而导致的数据库连接风暴，因为这种数据库连接避免了频繁启动和关闭服务进程带来的性能问题。

共享服务器模式还用于数据库连接穿透 NAT 防火墙。我们经常会碰到这样的应用场合，客户端在防火墙外，而服务器在防火墙内，出于安全考虑，数据库服务器的服务 IP 通过 NAT 解释为外网的 IP，提供给防火墙外的客户端使用。不过 SQL*Net 连接在穿透防火墙时往往会出现问题，由于 SQL*Net 在连接时首先连接监听器，然后由 LISTNER 分配一个服务进程，再将连接重定向到服务进程。正是这种重定向操作，会导致客户端出现“TNS-12535”之类的错误。为了解决这个问题，Oracle 提供了一个技术，叫做 Connect Manager（CMAN），CMAN 是一个 SQL*Net 的代理。当外网的客户端要连接数据库服务器时，首先不连接监听，而是连接 CMAN，CMAN 作为代理，接受客户端的连接，然后将 SQL*Net 包进行转发。为了避免连接过程中出现的重定向问题，CMAN 采用了预先连接的连接池技术，CMAN 在启动时就建立了一个连接池，并和数据库的服务进程建立了连接。因此客户端通过 CMAN 连接数据库时，不需要建立新的网络连接，就可以穿透 NAT 的防火墙。为了提高 CMAN 的效率，CMAN 连接池使用了共享服务器技术，这样一个服务进程就可以为多个通过 CMAN 连接的客户端提供会话服务。

从上面的讨论中，我们了解了会话及其容器服务进程之间的关系。会话是依赖于其服务进程的，一旦服务进程出现故障，会话也将会出现故障。因此，如果想要终止某个会话正在进行的操作，有两种办法，一是杀死该会话，我们可以通过 ALTER SESSION KILL 命令杀死这个会话；此外，也可以直接杀死会话对应的服务进程。

第 2 章

理解 DB Cache

2

DB Cache 是 Oracle 数据库中对性能影响最大的组件，优化 DB Cache 也是一个 DBA 最基本的职责。搞过软件开发的人都知道，缓冲区是提高性能的有效机制，DB Cache 的存在主要是为了提高会话访问数据文件中数据的效率。

Oracle 的 DB Cache 机制与其数据存储结构关系十分紧密，Oracle 的数据在文件中是以 Block 为单位存放的，因此 DB Cache 和数据块紧密对应。在内存中，DB Cache 存储的就是 Block 的完整镜像。

对于一般的 DBA 而言，其实并不需要十分深入地了解 DB Cache 的内部算法，只要了解一些最为粗浅的原理性规则，就可以解决日常遇到的超过 90%的相关问题了，只有少量的问题会涉及 DB Cache 的算法。因此，老白在本节的开始只介绍一些普通 DBA 需要了解的 DB Cache 的基础概念，随后的小节中介绍的关于 DB Cache 的内部原理是专门写给希望深入了解 DB Cache 的读者的，这些较为晦涩的文字，你完全可以先跳过，等到遇到问题时再来做深入的研究。通过下面介绍的内容，读者能够了解到 DB Cache 一些最为关键的概念，在一般的性能优化中，这些概念就能够帮助你分析问题了。

对于一般的 DBA 来说，需要了解下面几个关于 DB Cache 的概念。

- DB Cache 是以 Block 为单位组织的缓冲区，不同的 Block Size 的数据块对应于不同的 DB Cache。从 Oracle 9*i* 开始，RDBMS 支持多种不同块大小的表空间，如果我们要使用它们，就必须为这种特殊的数据块大小的数据文件设置单独的缓冲池。而默认的 DB Cache、Keep Pool、Recycle Pool 只能用于默认块大小的表空间。
- 用户访问 DB Cache 的数据比访问磁盘上的数据，速度要快数十倍甚至上百倍，因此应用系统应该尽可能多地从 DB Cache 中访问数据。在大多数情况下，DB Cache 的命中率越高，访问性能就越好。
- 数据库刚刚启动时，DB Cache 中几乎没有用户数据的缓冲（除非在系统级触发器中做了事先加载），当会话访问数据库的表和索引时，首先会检查 DB Cache 中是否存在该数据，如果不存在，就会从数据文件中读取该数据块到 DB Cache，然后再从 DB Cache 中读取该数据。
- 定位 DB Cache 中的数据块是通过散列算法实现的，对于 DB Cache 来说，Oralce 为了提高数据块定位的速度，为其设计了一个 Hash 链结构，将整个 DB Cache 中正在使用的数

据块都放置到 Hash 链上，这个 Hash 链是由多个 Bucket 组成的多链结构，每个 Bucket 就是一条链的链头，从链头引出一条独立的双向链。Oracle 还设计了一个算法，通过数据文件的文件号和数据块的块号，进行一个散列运算，得到的散列值就是这个数据块所在的 Bucket 的号码。如果要查找某个数据块，通过散列算法，算出这个数据块所在链的链头 Bucket 的位置，就可以很快地找到链头，从链头的双向链表结构扫描下去，即可找到相关的数据块。Bucket 的数量是固定的，一旦数据库启动后就不会改变，Bucket 的数量由参数_DB_BLOCK_HASH_BUCKETS 确定，8*i* 的默认值是 2 × DB_BLOCK_BUFFERS，也就是 Buffer 数量的两倍，9*i* 的默认值是大于两倍 DB Block Buffer 数量的最小素数，10*g* 的默认值是大于两倍 DB Block Buffer 数量的最小的 2 的幂次的数值。虽然 Oracle 的每个版本对该参数的定义略有不同，但是有一点是不变的，就是数据库缓冲区的 Hash Chains 的 Bucket 的数量始终大于 DB Buffer 数量的两倍。虽然我们无法获得 Oracle 内部散列算法的详细细节，但是有一点是肯定的，Oracle 设计 Hash 链的基本思路是，每个链上有最少的 Buffer 数量，最佳的情况是每条链上只有一个 Buffer。

- 普通会话只读取和修改 DB Cache，不负责将脏数据写入磁盘，该操作一般是由 DBWR 后台进程来完成的。
- 如果某个会话要访问的数据当前已经被更改了，而这个会话需要访问该数据的前映像（Pre-Image），那么可以通过 UNDO 中保存的数据，结合当前的数据块，生成一个 Pre-Image 的数据块缓冲，这个数据块缓冲是针对同一个数据块的不同版本，因此它将和当前的数据块存放在相同的 Bucket 上，也就是同一条 Hash 链上。也就是说，在 DB Cache 中，同一个数据块可能存在多个版本，这些版本都存储在同一条 Hash 链上。
- 访问 Hash 链时，为了保证数据访问的一致性，通过 cache buffers chains 闩锁来保护 Hash 链的数据结构。因此如果出现 cache buffers chains 闩锁争用，那么一般来说都和 Hash 链的访问有关。如果碰到该闩锁等待十分严重，首先应该检查是否存在逻辑读（buffer get）十分高的 SQL，看看这些 SQL 的执行计划是否存在问题，是否对大表做了全表扫描。此外，较严重的热块冲突也会加大 cache buffers chains 闩锁的争用，这一点也需要注意。
- 除了 Hash 链外，DB Cache 另外一个十分重要的链是 LRU 链，如果某个会话需要分配一个数据块缓冲区用于存放一个新的数据时，就会从 LRU 链上查找。实际上，LRU 链是多条链的总称，包含 LRU、LRU-W 和 LRU-AUX 等链，不过对于大多数 DBA 来说，不需要很深入地了解 LRU 的内部原理，只要知道 LRU 链是一种以类似 LRU 算法组织的链，最近不被使用的缓冲区会被最先重用即可。确实，早期的 DB Cache 在 LRU 上是会移动的，常用的缓冲会被换到 LRU 的热端，不常用的缓冲会被挤到 LRU 的冷端，一般来说会话分配 Cache 时，会从 LRU 的冷端开始查找。在冷端找到脏块就把脏块移到 LRU-W（DBWR 的任务就是将 LRU-W 中的脏数据写入文件，然后将这些缓冲区从 LRU-W 上释放，变为可用缓冲），找到没有被 PIN 住的非脏块，就完成了缓冲区的分配操作。
- 从 Oracle 8*i* 开始，LRU 的算法有所改进，LRU 链上的缓冲不再需要移动了，而是通过 tch 计数器来判断某个数据块是否为热块。改进后的算法，当会话在 LRU 上分配缓冲时，还

是从 LRU 的冷端开始查找，如果查到的某个非脏的缓冲区是热块（根据 tch 值的高低判断是否为热块），就会跳过这个热块继续往下查找。

- LRU 链通过 cache buffers lru chains 来保护，因此针对 LRU 链的操作都要使用该闩锁。如果该闩锁发生较为严重的争用，说明数据块缓冲的分配操作出现了性能瓶颈，大多数情况下，DB Cache 容量不足都可能导致该闩锁严重争用。另外，如果数据库缓冲出现了严重的抖动（比如，某个 SQL 扫描了一张大表，几个 G 的数据块被读入缓冲，不过这些缓冲只使用一次，很快就会被其他数据换出，如果经常出现类似的情况，数据库缓冲的抖动就会很严重），也可能导致该闩锁争用的加剧。
- Oracle 的多缓冲技术可以有效地避免数据库缓冲区的抖动，除了默认的数据块缓冲池（Default Pool）外，还可以将经常访问的数据放入 Keep Pool，一次性使用的数据可以使用 Recycle Pool。设置 Recycle Pool 需要应用开发配合，因此目前使用并不是很多，Keep Pool 的使用确实可以改善 LRU 算法的性能，适合几乎所有系统。
- RAC 环境下的缓冲区融合技术是一把双刃剑，缓冲区融合技术解决了多个实例安全访问同一个数据库的问题，同时也大大提升了全局缓冲的访问性能，但是如果全局缓冲传输操作的数量过大，很容易导致性能问题。因此在 RAC 环境下，更需要减少热块冲突，减少不必要的节点间 DB Cache 的传输。在 RAC 环境中，要提高全局缓冲的性能，除了在硬件上提高私有网络的带宽外，尽可能提高 DB Cache 的命中率也十分关键，DB Cache 命中率高了，可以减少大量由于 Cache 未命中而导致的实例间消息的数量，使 LMS 进程有更多的时间处理真正的 Global Cache。实际上 RAC 环境的缓冲区融合（Cache Fusion）技术是一种十分优秀的群集技术，通过高速的私有网络实现全局资源的共享，其共享效率比磁盘共享要快十倍以上。一个典型的磁盘单块读的响应时间约为 4 毫秒，如果要通过磁盘共享数据，一个操作包含了多个文件 I/O 操作（写、读），而通过缓冲区融合算法，只需要将 Cache 从一个节点传输到另外一个节点就行了，Cache 在高速的千兆网络上传输的时间一般来说小于 1 毫秒。

DB Cache 的算法十分复杂，我们没有必要去完全了解，只需要了解 DB Cache 的大体结构和总体算法。了解这些算法的目的是为了更好地理解相关的闩锁和等待事件，以便于分析和处理相关的性能问题。如果我们一味地去强调精确的算法，那么就本末倒置了。在本节中，老白将由浅入深地介绍 DB Cache，让大家掌握 DB Cache 性能分析和优化的基本技术。

2.1　什么是 DB Cache

DB Cache 是十分复杂的，很多 DBA 都想了解 DB Cache 到底是怎么构成的。事实上从 Oracle 发展了那么多年来看，DB Cache 的变化是很大的，Oracle 的每个大版本都会对 DB Cache 的相关算法做相当大的调整，从而大幅提升数据库的性能。不过虽然 DB Cache 的管理算法一直在变化，但其基本原理并没有发生太大的变化。在本节中，我们就简单探讨一下 DB Cache 的一些基本结构和原理，但本文并不是 DB Cache 的完整描述，只是通过一些知识点，让 DBA 了解 DB Cache

的基本面貌和算法。

首先，DB Cache 是由一系列共享内存组成的，是在 SGA 中统一分配的一个组件。SGA 管理的共享内存区域是通过类似 shmget,shmat 等 UNIX 系统调用获取的。根据操作系统及其参数设置的不同，SGA 可能是经过多个共享内存申请获取到的内存的总和。在 SGA 中，系统根据初始化参数的设置，分配相应的 DB Cache。

众所周知，SGA 是以 GRANULE 为单位的，大多数系统的 GRANULE 的大小一般为 16M，SGA 比较小的系统可能为 4M。每个 GRANULE 中包含多个 Buffer，这些 Buffer 的大小和数据块的大小相同。大家要注意的是，在 DB Cache 中，不仅包含了 Buffer，还有一种被称为 Buffer Head(BH)的控制结构，这种控制结构在 Oracle 的内部被命名为 kcbbh。各个版本的 Oracle 数据库的 kcbbh 结构都有所不同，以 Oracle 9*i* 为例，kcbbh 是一个 188 字节的控制结构，我们可以通过伪代码来看看它的详细结构，老白将通过下列伪代码结构来说明 DB Cache Buffer Head 的结构，如代码清单 2-1 所示。

代码清单 2-1

```
struct kcbbh {
  void *                HASH 链的指针;
  int                   表空间号;
  krdba                 DBA 号;
  ub4                   flag; /*  修改时需要 cache buffers chains 闩锁 */
  b1                    buffer 状态;
  b1                    buffer 持有状态; /*  NULL,SHR,EXCL  */
  word                  数据块的 class;
  kfil                  绝对文件号;
  kobjd                 对象的 data object ID ;
  kobjn                 对象的 object id;
  ptr_t                 buffer 的地址指针;
  kscn                  事务 DSCN;
  kgglk                 buffer 使用队列指针;
  kgglk                 buffer 等待队列指针;
  b1                    一个独立的修改调用产生的修改的数量;
  b1                    如果修改失败需要恢复的状态位;
  kgglk                 LRU 链指针;
  b1                    是否在 LRU-W 链上 kcbbhfoq; /*  TRUE 表示在 lru-w 上*/
  b1                    LRU 闩锁保护标识;
  ub2                   tch 计数器;
  ub4                   最近一次 tch 增长的时间;
  kcrda                 磁盘恢复需要的最低 rba;
  kcrda                 缓冲区恢复需要的最低 rba;
  kcbcr                 一致性度相关数据;
  kssob *               恢复相关 STATE OBJECT 的指针; /* 恢复时需要的结构  */
  kcrfkd                数据块中变化的最大 SCN;
  ub2                   延迟块清除计数;
  ub2                   kcbbhssid;
  ub2                   WORKING SET 中的 ckpt queue 的号码;
  kgglk                 ckpt queue 的指针;
  kgglk                 文件 ckpt queue 指针;
  struct kcbwds *       WORKING SET 的指针;
  struct kcbbh.UNK_lch_kcbbhsh UNK_lch_kcbbhsh;
}
```

从上面的伪代码中，我们可以看到在 DB Cache 中有一些很重要的链，这些链上串联的每个节点都是 Buffer Header，而不是直接的 Buffer，由 Buffer Header 指向真正的 Buffer。Oracle 的 Buffer 是由大小相同的数据块组成的。同一个 BUFFER 的所有数据块的大小都相同。Default Pool、Keep Pool、Recycle Pool 的大小和 DB_BLOCK_SIZE 相同。*n*k Pool 的 Buffer 的块大小和 DB_BLOCK_SIZE 不同，如图 2-1 所示。

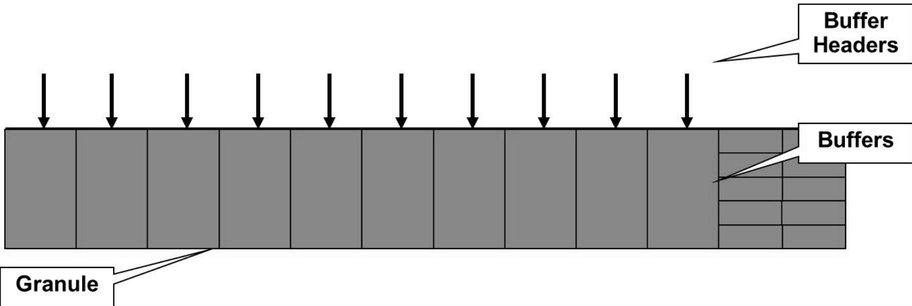


图　2-1

一个 Buffer 的 Buffer Head 处于同一个 GRANULE 中，在每个 GRANULE 的头部都是 Buffer Head，Buffer Head 的 kcbbhba 指针指向 Buffer 的地址。

对于不同大小的 Block 和 GRANULE，每个 GRANULE 中可以存放的 Buffer 数量不同，表 2-1 是 Julian Dyke 在其 *SGA Internals* 讲义中提供的参考数据。

表　2-1

数据块大小（字节）	每个Granule的BUFFER数量	
	4mb	16mb
2048	1875	7503
4096	979	3916
8192	500	2002
16384	253	1012
32768	127	509

在 Oracle 9*i* 数据库中，如果我们的 Block 大小是 8K，而 GRANULE 的大小为 16M，那么每个 GRANULE 可以包含 2002 个 Buffer，这和 16M 除以 8380（即 8192+188）得到的结果是一致的。

DB Cache 中有多种类型的链，这些链大多数是双向链表，这些链表可以归结为两类，LRU 链和 HASH 链。LRU 链根据 LRU 算法对 DB Cache 进行 Buffer 分配和换出（Age Out）的管理（从 Oracle 8*i* 开始，LRU 算法有所变化，在后面会详细描述），而 HASH 链能够加快对 DB Cache 的访问（大家要注意，这两种链表的作用是不同的，一个是为了分配和换出，另一个则是为了逻辑读）。

DB Cache 的算法从 8*i* 开始，有了很大的变化，因此我们要密切关注这些变化。在 8*i* 前，LRU 链中的 Buffer Head 是要根据“热”的程度在 LRU 链上移动的，一旦某个 Buffer 从“冷块”变为“热块”，那么这个 Buffer 就需要移到 LRU 链的热端，这样的算法虽然可以保证“热块”尽可能长时间地保存在 DB Cache 中不被换出，但是会增加 LRU 链的维护成本和相关闩锁的争用，从而降低 LRU 链的性能。从 Oracle 8*i* 开始，为了解决热块在 LRU 链上不停移动带来的性能问题，BUFFER HEAD 的数据结构中引入了一个新的计数器 TCH（访问计数器），这个计数器用于记录某个 Buffer 被访问的次数。由于 TCH 的引入，Buffer Header 不需要再在 LRU 链上移动了，当 TCH 达到某个阈值（通过参数_db_aging_hot_criteria 确定），那么这个块就被设置 Buffer 热块标识。当我们需要分配一个 Buffer 的时候，会话总是从 LRU 链的冷端（也叫尾端）开始向前搜索，在这种算法中，一个热块可能被挂在 LRU 链的尾端，但是因为我们引入了 TCH 和热块标识，因此当会话找到了一个可以使用的设置了热块标识的 Buffer 时，就会自动跳过这个 Buffer，继续往下搜索，而不会直接使用这个 Buffer，这样也就确保了访问比较频繁的数据块不会被很快换出。

新的算法也从另外的角度确保了“最近使用，最后换出”的原则。当一个新的数据块被载入 Buffer 时，系统会从 LRU 链上找到一个 Buffer（其实是找到 Buffer Header），然后摘下该 Buffer，写入数据，这时需要把这个 Buffer 再次挂入 LRU 链，这个挂入点是由参数_db_percent_hot_default 确定的，其含义是 LRU 链热端尾部的位置，这个参数是百分比，默认值为 50%。也就是说，新的 Buffer 将被挂入 LRU 链的中部，而老的 Buffer 将被一直往尾部挤。一般来说，这个参数的默认值对于绝大部分系统是不需要调整的，我们也可以通过调整这个参数，使新 Buffer 插入的位置更靠近热端或者冷端。调整这个参数对 LRU 算法的影响很大，因此要特别谨慎，对于 99%甚至 99.99%的系统来说，Oracle 的默认配置是足够好的，哪怕不是足够好，绝大多数 DBA 也无法计算出真正适合你的系统的最佳值。盲目调整这个参数可能导致十分严重的后果，请大家不要轻易在生产系统上进行尝试。

我们总在讨论 LRU 链，而实际上 LRU 链是一组链的总称，而不是一条链，在 Oracle 的各个版本中，LRU 链的数量是不一样的。从 8*i* 或者以后的版本来看，有以下几条重要的 LRU 链：

- ❑ LRU LIST（也叫 replacement list），前台进程从该链中查找可重用的 Buffer，这条链从 8*i* 开始不再采用 LRU 算法进行管理了，而采用了一种我们上面提到的改良的 LRU 算法，其核心部分就是 TCH。当一个新的 Buffer 加入这条链时，是加入到热端的尾部，热端尾部的位置由_db_percent_hot_default、_db_percent_hot_keep 和_db_percen\t_hot_recycle 这些参数确定，它们分别作用于 Default Pool、Keep Pool 和 Recycle Pool，_db_percent_hot_default 参数的默认值是 50%，也就是链的中间，_db_percent_hot_keep 和_db_percent_hot_recycle 是 0%。查找可重用 Buffer 时，从 LRU 的冷端开始，如果发现某个 Buffer 是热块，就会跳过。
- ❑ LRU-W LIST（write list），也就是我们常说的脏数据链（Dirty List）。LRU-W 存放的都是脏块，也就是需要存盘的数据块，DBWR 通过 LRU-W 生成写批量，完成数据块刷入硬盘的操作。

- LRU-AUX LIST 是 LRU LIST 中的一条子链，它的存在是为了提高 LRU 算法的性能。当 LRU-W 上的脏数据被写盘后，这些 Buffer 就可以重用了。但它们并没有被马上放入 LRU 链，而是被放入 LRU-AUX 链。前台进程查找空闲 Buffer 时，首先从 AUX 链开始查找，只有在 AUX 链中找不到可用的 Buffer 时，才去搜索 LRU 链。
- LRU-P LIST 这条链上的对象，一般在被写入时都存在一定的锁或者被某个会话 PIN 住了，因此这些 Buffer 往往被写入后还不能马上使用，需要释放锁或者降低锁的级别。这条链上的 Buffer 在尾部加入，DBWR 从头部开始写，一旦某个 Buffer 处于可释放的状态，DBWR 会将其移到 LRU-AUX。
- LRU-XO LIST（reuse object list），当某个对象被刷新（DROP、TRUNCATE 等）时，CKPT 进程将这个对象的所有 Dirty 和 Current 的 Buffer 放入 LRU-XO LIST。
- LRU-XR LIST（reuse range list），类似于 LRU-XO LIST。

全表扫描会对 DB Cache 造成较大的影响，因为有大量的数据块需要被扫描，并放入 DB Cache 中。为了尽可能少地影响 DB Cache，对于大表的全表扫描，Oracle 数据库设计了一个算法，使这类扫描操作增加的 Buffer 被放在 LRU 链的尾部，而不是中部，因此这些 Buffer 会被最快换出，尽可能不影响 LRU 链中的热块，不会将较常用的数据很快挤到尾部。而小表的全表扫描是和大表全表扫描不同的，小表全表扫描对 DB Cache 的影响较小，因此可以对这些 Buffer 采用和普通 Buffer 一样的操作，而不是将它们放到 LRU 链的尾部。

前面讨论了关于 LRU 链的相关概念和基本算法，大家也了解到了 LRU 链主要用来分配 Buffer 和管理 Buffer 的换出。实际上，对于 DB Cache，还有一个问题需要关注，就是如何对 Buffer 进行寻址。在一个很大的数据库缓冲区中，我们如何通过最小的代价找到相关的 Buffer，这一点十分关键。因为在 Oracle 数据库中，如果我们要查找某个数据，首先需要到 DB Cache 中去查找；如果在 DB Cache 中找不到，我们才会从物理文件中读取。在一个并发量很大的系统中，这种寻址的效率决定了系统的性能。当我们要访问某个数据块的时候，能够依赖的参数就是文件号和块号。它们决定了某个特定的数据块，因此我们可以使用文件号和块号作为寻址的参数。Oracle 数据库的 HASH 链就是基于这个思路设计的。Oracle 在共享池中设计了一个物理结构，就是 HASH 链结构，DB Cache 的 HASH 链由多条子 HASH 链组成，每条子链我们称之为一个 Bucket。每条 HASH 链都是一条双向的链表，链表的每个节点对应一个 Buffer Header。默认情况下，在 HASH 链结构中，Bucket 的数量大于 Buffer 数量的两倍，一个数据块的文件号和块号通过散列算法，会定位到唯一的一个 Bucket 上，因此该数据块在 DB Cache 中的 Bucket 的位置是固定的。通过这种机制，Oracle 就可以在 DB Cache 中快速定位到这个数据块了。由于采用的是散列算法，因此不同的数据块可能得到相同的散列值，也就是说它们存放的 Bucket 可能是相同的，那么这些 BUFFER 就被串在同样的 HASH 链上。不过这种情况出现的机会不大，一条 HASH 链上的节点数量一般都很少，因此查找某个数据块的 BUFFER 的操作一般都是很快的。

散列算法使用的参数是文件号和块号，因此一个数据块的不同版本，在 DB Cache 的 HASH 链中肯定在同一条链上，因为它们具有相同的文件号和块号。之前我们也了解过，一个数据块的一致性读块（CR Block）是由当前状态的数据块和 UNDO 中的相关数据合并后生成的，能够反

映之前某个时间点的数据情况。在一个变化十分频繁，大查询（持续时间较长的查询）较多的系统中，某个数据块的 CR Block 的数量可能会很多，这样就会增加查找该数据块的 CR Block 的操作开销，在操作过程中持有闩锁的时间也会变长。

综合我们前面讨论的 LRU 链和 HASH 链，我们用图 2-2 来描述 DB Cache，这只是一张示意图，真正的结构要复杂得多，这里忽略了很多 DB Cache 的细节，不过从图中可以了解到 DB Cache 的概貌。

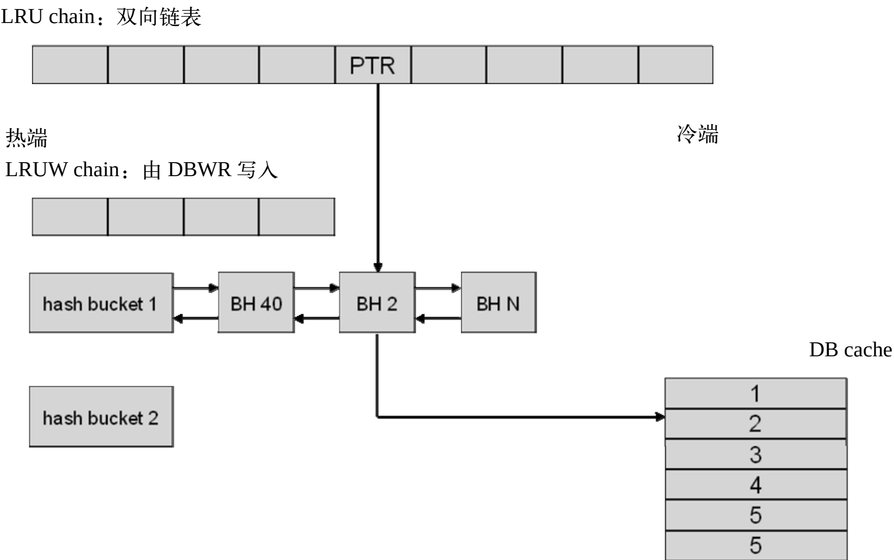


图　2-2

如果要真正了解 DB Cache 的结构，首先需要了解一下 Working Set 的概念。数据结构 kcbwds 是 Kernal Cache Buffer Working Set management Descriptors 的简称，在数据库中 X$KCBWDS 视图就是内存结构 kcbwds 的关系型描述。我们可以通过伪代码看一下 10.2 版本数据库的 kcbwds，如代码清单 2-2 所示。

代码清单 2-2

```
struct kcbwds {
  kcbkcql           ckpt q 闩锁[2]; /*  2 个闩锁  */
  kcbsds            WS 虚拟扩展;
  ub4               WS 的状态;
  kcbwlid           WS ID;/* 对应 bh 中的 WORKING SET ID  */
  word              包含这个 WS 的 BUFFER POOL;
  word              WS 的块大小索引号;
  dblkz             WS 的块大小;
  word              WS 的 NUMA 处理器租;
  word              WS 使用的 DBWR 号;
  kcbwl             WS 闩锁;
```

```
  kcbwbl                WS 列表; /* 包含 LRU 和 LRU-AUX 链的指针等信息*/
  kgglk *               LRU 链上的冷端第一个 BH 的地址;
  uword                 LRU 链上热块的百分比;
  uword                 LRU 链上热端最多的 BUFFER 数量;
  uword                 LRU 链上目前热端的 BUFFER 数量;
  uword                 当前 WS 中的 BUFFER 数量;
  kcbohtab *            object queue 的 HASH 表的指针;
  uword                 前台会话从 WS 中读取 buffer 的计数器;
  word                  空 buffer 申请的计数器;
  uword                 dbwr 从这个 WS 写的 buffer 的计数器;
  uword                 dbwr 在 WS 中扫描的 buffer 的计数器;
  uword                 在 WS 中产生 free buffer waits 的计数器;
  uword                 在 WS 中产生的 write complete 等待的计数器;
  uword                 在 WS 中产生的 BUFFER BUSY WAITS 的计数器;
  uword                 在 WS 中产生的 Free Buffers Inspected 的计数器;
  uword                 在 WS 中产生的 Dirty Buffers Inspected 的计数器;
  uword                 在 WS 中产生的 Pinned Buffers Inspected 的计数器;
  uword                 在 WS 中产生的 Hot buffers moved to head 的计数器;
  uword                 在 WS 中产生的 DB Block Changes 的计数器;
  uword                 在 WS 中产生的 DB Block gets 的计数器;
  uword                 在 WS 中产生的 Consistent gets 的计数器;
  uword                 在 WS 中产生的物理读的计数器;
  uword                 在 WS 中产生的物理写的计数器;
  uword                 WS 中前台进程扫描深度;
  uword                 脏数据最大比例（超过该比例激发 DBWR 写）;
  uword                 pwbcnt;
  uword                 protcnt;
}
```

如果你认真地阅读了上面的伪代码，就不难发现，Working Set 实际上定义了一个 DB Cache 的工作组。Oracle 为了提高 DB Cache 的刷新性能，将整个 DB Cache 划分为多个 Working Set。每个 Working Set 管理一部分 DB Cache，且拥有独立的 LRU 链结构（包含 LRU、LRU-W、LRU-AUX 等），并有独立的闩锁进行管理。

在这段伪代码的后半部分，包含了这个 Working Set 的一些统计信息，我们可以在 AWR 报告中看到这些统计信息。

不同的 Working Set，是由独立的 DBWR 进行维护的，一个 DBWR 进程管理一个或者多个 Working Set。在多 CPU 环境下，设置多个 DBWR，用于管理多个 Working Set，这样可以提高 DB Cache 脏块的处理速度。如果我们碰到了大量的 free buffer wait 等待或者 write complete 等待，那么说明可能是 DB Cache 太小了，或者 DBWR 写入速度太慢了，这时增加 DBWR 的数量也许可以有效地改善这方面的性能。不过一般情况下，将 DBWR 的数量设置得过多（比如大大超过 CPU 数量）意义并不大，因为这样可能会导致 CPU 资源出现争用，无法保证所有的 DBWR 都能够有足够的 CPU 资源进行工作。这时可能有些朋友就会考虑了，是不是 DBWR 数量大于 CPU 的数量就没有意义了呢？实际上并没有那么简单，这是一个需要综合考虑的问题，因为 DBWR 采用的是批处理写的模式，组织写作业和异步写这些操作可以交叉进行。如果系统的 CPU 资源还未碰到瓶颈，设置 DBWR 数量超过 CPU 的数量也可能提高 DBWR 的总体性能。不过一般情况下，如果脏块产生过快，说明系统的负载很高，这时候设置太多的 DBWR 帮助不大。

2.2　DB Cache 的分配和 DBWR 的相关算法

在 DB Cache 的算法中，数据库启动之初，空闲的 Cache 都是在 LRU 链上的。前台进程需要分配 DB Cache 的时候，从 LRU 链的冷端开始扫描，查找可用的 Cache。找到 Cache 后，根据要读入的数据块的 RDBA，通过散列算法找到一个 HASH 链，将这个数据块头上的 HASH 链相关指针链接到 HASH 链上。

前台进程只负责将数据块从文件中读取到 DB Cache 中，而 DB Cache 中的脏数据是由专门的后台进程来负责写入数据文件的，这个后台进程就是 DBWR。在介绍详细的算法之前，我们首先需要学习几个数据库的统计数据。这几个统计数据对于我们理解 DB Cache 的算法十分重要。由于 DB Cache 的算法改进十分频繁，因此本节介绍的一些算法，都是基于 8*i* 版本的，9*i* 或者更高的版本中，算法的核心并未改变，只是在细节上有所不同，请大家阅读本节时注意。

首先我们来看看 Foreground scan depth 这个参数，它和前台进程搜索 DB Cache 有关。前台进程要分配 DB Cache 的时候，会从 LRU 的尾部开始搜索，如果发现了可用的 DB Cache 就 PIN 住使用。如果找到的这个 Cache 目前是脏的，还没写盘或者被其他会话 PIN 住，或者被标识为热块，那么这个 Cache 就不能使用，需要继续向前搜索。而这种搜索工作有可能进行很长时间都无法找到可用的 Cache，这样就极大地影响了 LRU 链的性能，因此需要定义一个最大深度，一旦超过这个搜索深度，就放弃搜索，同时记录一个标志位，通知 DBWR 刷新脏数据到磁盘。同时前台进程进入休眠状态，等待 free buffer 的消息，收到该消息后重新开始查找工作。这个深度默认设置为 DB Cache 的 1/4，并可以通过参数_db_block_max_scan_pct 来调整（在每个版本中，该参数的初始值都会有所不同）。

在搜索可用 Cache 的时候，如果发现某个 BUFFER 是脏的，那么前台进程就会马上将该 BUFFER 从 LRU 链上摘下，放到 LRU-W 的尾部，并且增加 buffers moved to the dirty queue by foregrounds 统计量。如果发现 LRU-W 链的长度已经大于 Max dirty queue，那么前台进程也会停止搜索操作，并记录一个标志位，通知 DBWR 将脏数据写到磁盘上。同时前台进程进入休眠状态，等待 DBWR 完成写入。这种情况下，如果 known clean buffers 计数器的值不是 0，那么就会减少 known clean buffers 的计数。known clean buffers 是一个计数器，该计数器记录的是当前的 DB Cache 中可能的干净块（也就是非脏的块）的数量。这个计数器在 DB Cache 相关算法中还有一些其他的用处，等老白慢慢介绍。Max dirty queue 的大小是 Max write batch 的两倍，Max write batch 是一次写批处理操作的最大数量，可以通过参数_db_block_write_batch 来调整。

如果前台进程在搜索中找到了可用的 BUFFER，那么它就会将该 BUFFER 移动到 LRU 链热端的尾部，同时减少 known clean buffers 的计数。如果 known clean buffers 计数小于 dbwr scan depth 的一半，那么前台进程也会设置标识，触发 DBWR 写操作。一般情况下，dbwr scan depth 初始化时的值等于 min scan depth（默认为 DB Cache 的 1/4），不过这个参数会在数据库运行过程中动态调整。

上面所说的算法并不能完全反映 Oracle 的真实算法，不过大体上能够体现其主要思想。当 DB Cache 中可用的空闲块不足的时候，会设置 DBWR Make Free Message，该消息会唤醒休眠的

DBWR 进程，并开始工作。当这个消息被 DBWR 收到后会发生什么呢?

DBWR 首先会获取 LRU 闩锁， 清除 DBWR Make Free Message 标志，然后会检查 LRU-W 链。如果 LRU-W 是空的，并且 known free buffers 计数大于 dbwr scan depth 的一半，那么 DBWR 什么都不做，直接释放闩锁，然后继续“睡觉”。

如果 DBWR 发现 LRU-W 上的 BUFFER 数量不够一次 DBWR BATCH 操作，那么 DBWR 就会从 LRU 的尾部开始搜索脏数据块，发现后，将这个数据块移动到 LRU-W 的尾部。如果扫描的数据块没有被移动到 LRU-W，那么 clean buffers scanned 计数器会增加。搜索结束的条件是 LRU-W 上的数据块数量达到了 dbwr batch 的数量，或者搜索的深度达到了 dbwr scan depth 的值（该参数在数据库运行过程中会自动调整，调整范围在 min scan depth 和 max scan depth 之间）。扫描结束后，known clean buffer 的值被重置为 DBWR 扫描后的 clean buffers scanned 的值与本次 dbwr write batch 写入的缓冲区数的总和。此时，LRU 闩锁被释放，然后 DBWR 将脏数据块写入数据文件，并将已经写入的缓冲区放入 LRU-AUX 链。

如果 DBWR 写入完成后，LRU-W 链空了，或者 known clean buffer count 已经大于 dbwr scan depth 的 3/4，那么 dbwr scan depth 将被减少一个增量。因为系统认为 DBWR 的写操作总量可以略微降低。如果写入后，known clean buffer count 还小于 dbwr scan depth 的 1/2，那么 dbwr scan depth 会增加一个增量，因为系统认为 DBWR 的写操作总量需要增加。

除了 Make Free Message 外，还有其他的消息可以唤醒 DBWR，最为典型的是 Checkpoint 消息和 Reuse Block Range 消息。Reuse Block Range 可能由 drop、truncate 等 ddl 命令发出。此外每隔 3 秒钟，会产生一个 DBWR timeout 事件，如果 DBWR 发现休眠超时后，LRU-W 还是空的，那么它就会以两倍于 dbwr scan depth 的深度搜索 LRU-W，寻找脏数据块。

本节介绍了 DBWR 的一些基本算法，这些算法是以 Oracle 8*i* 的基础版本为对象的，其实我们学习这些算法的目的并不是为了写一套新的 RDBMS 管理软件，而是从中理解一些 DB Cache 和 DBWR 的配置、优化、维护方面的思路。在随后的几节中，我们将逐步展开讨论这些内容。

2.2.1 DB_WRITER_PROCESSES 参数

对于绝大多数系统来说，只要 I/O 能力足够强，并且 DB Cache 容量足够大，那么在一般情况下，单个 DBWR 进程基本上是能够胜任的。不过对于 I/O 十分敏感的系统或者并发量十分巨大的系统，使用多个 DBWR 进程有助于提高脏数据块写的性能。Oracle 7 虽然也支持多个 DB WRITER 进程，但是这些进程之间是有区别的，其中一个是 MASTER，其他几个是 SLAVER。SLAVER 进程是不能使用异步 I/O 的，只能使用模拟的异步 I/O。从 Oracle 8.0.4 开始，DBWR 的算法发生了革命性的变化。DBA 可以使用两种策略来配置 DBWR，一种是单个 DBWR 进程加上多个 I/O SLAVER 进程；另外一种是配置多个 DBWR 进程。不论是使用 DBWR 还是 I/O SLAVER，都可使用异步 I/O。这种改进大大提高了数据库写 I/O 的性能。

在使用 Oracle 8 以上的版本时，如何配置 DBWR？是使用多个 DB WRITER 进程，还是一个 DB WRITER 进程加上多个 IO SLAVER 进程呢？实际上这两种配置方法很难分出优劣，在不同的系统上，可能两种配置的效果差不多，也可能有所区别。具体使用哪种方式，只有做过实验才能

得到准确的结论。十年前，关于这两种方法的优劣，在网上有过一场论战，最终论战双方也都无法说服对方。实际上，在当时的系统配置情况下，这两种方式的主要区别在于，如果使用单一的DBWR 进程加上多个 I/O SLAVER 进程，那么这种配置和 Oracle 7 的多 DBWR 进程的机制是类似的，不同的只是 SLAVER 进程支持异步 I/O。由 MASTER 的 DBWR 进程来管理 LRU 链和组织 WRITE BATCH，而实际的 I/O 操作是由 SLAVER 进程完成的，MASTER 和 SLAVER 之间采用类似于异步 I/O 的机制，将写任务交给 SLAVER 后，MASTER 可以继续进行下面的操作，从而提高 DBWR 的处理能力。

如果使用多个 DBWR 进程，那么每个 DBWR 管理 LRU 链的一个子区域。每个 DBWR 都有自己的 LRU LATCH，对于 Oracle 8 和 Oracle 8*i*，LRU LATCH 通过参数 DB_BLOCK_LRU_LATCHES 来设置（Oracle 9*i* 以后的版本中取消了这个参数，自动将每个 DB Cache 的 LRU LATCH 数量设置为 CPU 数量的一半）。

DB_BLOCK_LRU_LATCHES 定义了所有 DB Cache 的 LRU LATCH 总数。一般来说，建议将该参数设置为不超过 CPU 的数量（对 Oracle 8*i*，DB_BLOCK_LRU_LATCHES 的最大值为 CPU 数量的 6 倍）。而一般情况下，DBWR 的数量也不应该超过 DB_BLOCK_LRU_LATCHES 的数量，否则会形成 LRU 闩锁的竞争。Oracle 8/8*i* 对于使用多缓冲的情况，除了 DEFAULT POOL 以外，每个 POOL 的 LATCH 数量默认为 1，也可以在 DB_BUFFER 参数中指定 LATCH 的数量，DEFAULT POOL 的 LATCH 数量是总数减去其他 BUFFER 的 LATCH 数量。

随着硬件技术的发展，系统拥有了更多的 CPU、更大的内存和更快的 I/O 子系统。如今，配置 DBWR 不仅仅要考虑 I/O 的能力，因为现在的 DB Cache 已经不像十多年前那样，几个 G 就算是超大型的系统了，而是动辄几十 G 甚至几百 G。LRU 的管理性能成为一个十分重要的考虑因素，因此由单个 DBWR 来管理 LRU 显然是不合适的，使用多个 DBWR 进程成为主要的选择。

基于上面关于 DBWR 的基础知识，在设置 DBWR 数量的时候，我们可以如此考虑：

- 一般情况下，如果没有发现明显的 DBWR 问题，那么不需要使用多个 DBWR。
- 如果确实需要使用多个 DBWR，那么对于多 CPU 系统，一般可以直接使用；对于单 CPU 系统，可以采用一个 DBWR 加上多个 I/O SLAVER 的方式。这条经验不是绝对的，为了确保获得最佳的性能，最好是两种方式都尝试一下。
- 如果你使用的是 9*i* 或者更高的版本，对于 I/O 不存在严重瓶颈的系统，那么就毫不犹豫地使用多个 DBWR 吧。
- 使用多个 DBWR 时，建议 DBWR 的数量不要超过 CPU 的数量（对于 Oracle 8/8*i*，DBWR 的数量不超过 DB_BLOCK_LRU_LATCHES）。

那么我们如何来判断目前的 DBWR 是否足够呢？STATSPACK 报告或者 AWR 报告是十分好的工具。

```
                                                        Free Writ     Buffer
     Number of Pool         Buffer     Physical    Physical Buff Comp       Busy
P      Buffers Hit%           Gets        Reads      Writes Wait Wait      Waits
--- ---------- ---- -------------- ------------ ----------- ---- ---- ----------
D    4,405,998   98    945,225,805   21,899,493   2,291,122    0    0  5,692,101
```

如果经常会有 write complete waits 存在，那么就说明 DBWR 的写性能不足，可以考虑通过增加 DBWR 来改善脏块写的性能。另外，在出现 free buffer waits 的问题时，如果我们无法通过扩大 DB Cache 来改善其性能，那么就需要增加 DBWR 的数量，这样也可以改善这方面的问题。

最后要强调的一点是，不要误解了本节的观点，由于本节介绍的内容是从 Oracle 7 到现在的 11.2 各个版本的情况，因此有些观点可能更适合于早期的 Oracle 版本。在 10*g* 或者 11*g* 中，大家不需要花费太多的精力纠结于 DBWR 的配置，大多数情况下，默认的配置就没有问题。只有少数情况，我们需要增加 DBWR 的数量。调整 DBWR 的数量是很专业的操作，一定要在深思熟虑后再去进行。

2.2.2　DB Cache 的几个主要的链和 CKPT 算法

DB Cache 中有很多重要的链，这些链或者和 DB Cache 的分配、换出、刷新优化有关，或者和 DB BLOCK 的查找有关。下面我们先来了解这些链的基本情况，然后再来探讨它们在各种 DB Cache 活动中所起的作用。

1. LRU list 和 LRU-AUX list

LRU list 也叫 replacement list，就是大家常说的 LRU 链，我们在前面的章节已经做了很详细的介绍，这里就不多说了。要特别提到的是 LRU-AUX list，它是 LRU 链的一条子链。从 LRU-AUX 头开始的 BLOCK 是被确认为 CLEAR 的，其来源包括 DBWR 已经写回文件的数据块和 UNPIN 的干净块。前台进程查找空闲 BUFFER 的时候，会首先在 LRU-AUX 中查找，因为从这里找到的 BUFFER 一般都是可用的。

2. LRU-W list (write list)

LRU-W list 就是我们常说的脏数据链，在前面的章节中，我们已经详细介绍过，对于 LRU-W 链，也存在一个 AUX 链——LRUW-AUX。引入 AUX 链是为了提高 LRU-W 的效率，支持异步 DBWR 作业。在没有引入这条链以前，当一个 DBWR 作业还没有完成时，是无法进行下一个 DBWR 作业的。引入 AUX 后，AUX 链存放的是当前 DBWR 正在写入的数据，这样 LRU-W 就可以腾出来处理下一个 DBWR 作业了。

3. LRU-XO list (reuse object list)

LRU-XO list 也被称为重用对象链，这条链主要对 TRUNCATE、DROP 等操作对象相关的 BUFFER 进行 CHECKPOINT 操作。当 reuse object cross Instance call 事件发生的时候，CKPT 会将这个对象的脏数据块放入 LRU-XO 链，然后由 DBWR 将这些数据块写盘。当 CKPT 发现这条链已经为空时，本次 CHECKPOINT 操作宣告结束。

4. LRU-XR list (reuse range list)

LRU-XR list 也被称为 Reuse range 链，当 reuse range cross Instance call 事件发生的时候，CKPT 搜索这些 BUFFER，将脏数据块放入 LRU-XR 链，然后再由 DBWR 将这些数据写盘。当 CKPT 发现这条链已经为空时，本次 CHECKPOINT 宣告结束。如果我们要将某个表空间 OFFLINE 或者将其设置为 READ ONLY，此时就会产生 ruse range cross Instance call 事件，因为要想安全地完成这些操作，必须要将表空间上的脏数据全部写入磁盘。

上面介绍了一些 LRU 链，下面我们再来看看 BUFFER HEAD 的数据结构，了解一下这些链是如何使用的（代码清单 2-3 所示的伪代码是 10*g* 的 BH 的一部分，为了方便读者更清晰地查阅，这里去掉了一部分不相干字段）。

代码清单 2-3

```
kgglk                HASH 链的指针;
ktsn                 表空间号;
krdba                RDBA 地址;
kobjd                存储 OBJNO; /*  不一定有值  */
kobjn                字典 OBJNO; /*  不一定有值  */
ub2                  绝对文件号;
b1                   BUFFER 修改计数;
ub2                  仿真组 ID;
ub1                  在 WORKING SET 上的 CKPT 队列号;
kgglk                ckpt queue 的链接指针;
kgglk                文件 CKPT QUEUE 的指针; /*每个文件都有一个 ckpt queue，
                                             在 reuse range 时使用 */
ptr_t                指向真正的 BUFFER 的指针;
kgglk                正在使用该 BUFFER 的使用者的列表;
kgglk                正在等待该 BUFFER 的列表;
kgglk                kcbbhrpl;/* BUFFER 的 LRU 链指针*/
b1                   是否在 LRU-W 上; /*如果在 LRU-W 上，该值为 TRUE*/
b1                   LRU 闩锁保护标识;
ub2                  TCH;
void *               指向 TX STAT OBJECT 的指针;
kcocv *              指向该 BUFFER 的最后一个 CHANGE#的指针;
struct kcbwds *      WORKING SET 的指针;
krfgda               kcbbhgda; /*闪回数据的磁盘地址*/
kgglk                kcbbhoq; /*对象 CKPT QUEUE 的指针，在 REUSE OBJECT 时使用*/
```

根据上面的数据结构，接下来我们要推测 CKPT 操作了。以前经常有人问我 CKPT 的相关操作，我在这方面的研究较少，主要是因为相关的 Oracle 文档实在是太少了。虽然从 Tom 和 Steve 那里可以获得一些资料，但是都比较细碎，因此针对 CKPT 操作的一些细节问题，我也只能通过上面的数据结构做一个推测，这个推测不一定准确，不过大体上可以反映出 CKPT 算法的一些最基本的概念。

首先，我们来看看 CHECKPOINT QUEUE。它是通过 LOW RBA 排序的一个链表，当某个 BLOCK 发生变化的时候，会将该 BLOCK 按照 LOW RBA 地址链入相关的 CHECKPOINT QUEUE，这些链表包括标准 CKPT QUEUE（老白在这里用了“标准”二字，实际上这并不是 Oracle 内部的术语，这部分内容公布的很少，因此这里只是为了将其区别于 RANGE 和 OBJECT 这两个 CKPT QUEUE）、XO CKPT QUEUE（OBJECT CKPT，每个对象一个）和 XR CKPT QUEUE（RANGE CKPT，术语称为 PER-FILE CKPT QUEUE，每个数据文件一个）。

在进行普通的 CHECKPOINT 操作时，CKPT 进程先找到 CKPT 链的头部，将 CKPT QUEUE 交给 DBWR，再由 DBWR 组织 CHECKPOINT DBWR 批处理，然后 DBWR 进程开始写入操作，由于 CKPT QUEUE 是按照 LOW RBA 的顺序排序的，因此在做 CHECKPOINT DBWR 批处理时，要按照 RBA 的顺序写入脏数据。在 DBWR 完成写操作后，CKPT 进程更改控制文件和相关文件

头的 SCN 数据，记录本次 CKPT 操作的结果。

由于 CKPT QUEUE 上的 BUFFER 有可能在 LRU 链上，也有可能在 LRU-W 链上，在组织 DBWR 批处理时，LRU 链上的 BUFFER 是否要摘下放入 LRU-W？这些算法在 CHECKPOINT 写入的时候，BLOCK 不从 LRU LIST 上移走，写完后，从 CKPTQ 里清除，这和普通的 LRUW 中的块被写入数据库是不同的。

在进行 TRUNCATE、DROP 这类操作时，也会发生 CHECKPOINT，这类 CHECKPOINT 被称为 REUSE OBJECT CHECKPOINT，具体的操作细节目前很少有资料说明。老白猜测，当 REUSE OBJECT CHECKPOINT 发生时，CKPT 进程将 kcbbhoq 指向的 OBJECT CHECKPOINT 链上的脏块链入 LRU-XO，交给 DBWR 处理。DBWR 组织写批处理，进行写入操作。只有当所有的 LRU-XO 上的块全部被写入后，DDL 操作才能完成。这种算法在用 TRUNCATE 拼接一个大对象时可能会很慢（如果这个对象有很多脏块的话）。10*g* 的 kcbbhoq 字段是一个对象的 BUFFER 链。这条链的存在对于 reuse object cross Instance call 的性能有很大的帮助。

当表空间设置为 READ ONLY 或者 OFFLINE 的时候，会触发 REUSE RANGE 的 CHECKPOINT，其原理和 REUSE OBJECT 的 CHECKPOINT 类似。通过在 BUFFER HEAD 中的 kcbbhfql 将每个数据文件的脏块串成一条链。这样，当 REUSE RANGE CHECKPOINT 发生时，CKPT 将这条链链入 LRU-XR，然后交给 DBWR 处理。

2.2.3　检索某个 DB BLOCK 的模拟算法

很多朋友都在研究检索数据块的内部算法，不幸的是 Oracle 公司并没有公开这个算法，而且这是 Oracle 数据库性能的最核心技术之一，这部分算法是永远也不可能公开的。老白根据这些年对 Oracle 内部原理的研究，编写了一段伪代码，通过这段伪代码向各位读者展示一下 DB Cache 管理的一些算法细节。这段伪代码只能够模拟 DB Cache 管理的一些皮毛，在具体的实现细节上，Oracle 的内部原理要复杂得多。下面我们就来看看这段伪代码，如代码清单 2-4 所示。

代码清单 2-4

```
main()
{
int vRDBA;
//获取该数据块的HASH链的散列值
v_hash_value=getRdbaHash(vRDBA);
//SPIN 相关cache buffers chains,根据散列值选择不同的子闩锁
while (!spinLatch（'cache buffers chains',v_hash_value))
  {
  //如果spin失败，就sleep，然后继续SPIN
   pinLatchSleep(sleepTime++);
  }
//在指定的hash chains中查找符合某个SCN条件的数据块的BUFFER HEAD
retVal=findBufferInHashChains(v_hash_value,vRDBA,vScn,);

if(retVal==O_找到兼容版本数据)
//如果找到了兼容的版本的DB Cache数据，那么通过BH中的指针找到这个Cache的地址，
//然后将该地址写入UGA的访问该数据块的库缓存的访问数据块列表中
```

```
    {
        //vkcbh 是这个 buffer 的 buffer head，这个 bh 被链接在 HASH CHAINS 上，
            //kcbbhba 是 BH 对应的 BUFFER 的地址
        vksuse->ksusesql->...=vkcbh->kcbbhba;
        //释放 cache buffers chains 闩锁，然后返回，结束访问
        unpinLatch('cache buffers chains',v_hash_value);
        return;
    }
else
    {
        if(retVal==(O_找到 BUFFER||O_当前快||O_SCN 不兼容)
            //需要生成 CR BLOCK，然后读取
            {
        //释放 cache buffers chains 闩锁，然后返回，结束访问
            unpinLatch('cache buffers chains',v_hash_value);
        //查找一个空闲的 BUFFER，用于生成 CR block
           //查找空闲块，将起 buffer head 赋予 vbh，如果找不到，反复重试
           while((vbh=findFreeBuffer())==null)
           {
            //如果没有找到空闲块，发出 makefree 消息，然后 SLEEP，等待唤醒
              sendMakeFreeMsg();
              sleepForFreeMsg();
          }
      //再次获取相关 cache buffers chains,根据散列值选择不同的子闩锁
       while (!spinLatch（'cache buffers chains',v_hash_value))
        {
        //如果 spin 失败，就 sleep，然后继续 SPIN
         pinLatchSleep(sleepTime++);
        }
            //下面操作省略，算法十分复杂，先复制 CURRENT 的 BUFFER 到新 BUFFER，
                //然后通过 UNDO 数据前滚数据块的数据
            ...
            ...
            ...
            //处理完成,释放 cache buffers chains 闩锁，然后返回
            unpinLatch('cache buffers chains',v_hash_value);
        return;
            }
  }
  else
    //没有找到 BUFFER，找一个可用的 BUFFER，从数据文件读入相关 BLOCK
    {
        //释放 cache buffers chains 闩锁，然后返回，结束访问
            unpinLatch('cache buffers chains',v_hash_value);
        //查找一个空闲的 BUFFER，用于读取数据块
           //查找空闲块，将起 buffer head 赋予 vbh，如果找不到，反复重试
           while((vbh=findFreeBuffer())==null)
           {
            //如果没有找到空闲块，发出 makefree 消息，然后 SLEEP，等待唤醒
              sendMakeFreeMsg();
              sleepForFreeMsg();
          }
      //再次获取相关 cache buffers chains，根据散列值选择不同的子闩锁
       while (!spinLatch（'cache buffers chains',v_hash_value))
        {
        //如果 spin 失败，就 sleep，然后继续 SPIN
```

```
             pinLatchSleep(sleepTime++);
            }
            //下面省略，从数据文件中读取该数据块
            ...
            ...
            ...
              //处理完成，释放cache buffers chains闩锁，然后返回
              vksuse->ksusesql->...=vkcbh->kcbbhba;
              //以下省略，从数据块中查找所需要的记录
                ...
                ...
              unpinLatch('cache buffers chains',v_hash_value);
    return;
        }
    }
```

上面这段代码是老白随手涂鸦之作，可能和 Oracle 实际的算法相去甚远。不过它体现出了 Oracle DB Cache 操作的一些基本的原理性规则，大家可以参考。在访问某个数据块时，首先要计算出该数据块的散列值，然后获取该 HASH 链的相关闩锁，再去搜索 HASH CHAINS。搜索的时候不仅要找到 RDBA 一致的数据块，而且其 SCN 也要符合要求，如果 SCN 不符合要求，那么可能需要生成 CR 块，再从 CR 块中读取相关数据。

这部分的知识点十分丰富，而且和很多知识点都有关联和交叉，因此老白这段代码只是考虑了最为简单的场景。如果大家有兴趣，可以继续细化这段代码，从而更深入地理解这些知识点。

这部分相关的知识点包括：working set(x$kcbwds)、DB BLOCK（包括数据文件、表空间、extent、segment 等）、buffer head(x$bh)、hash chains、lru chains、undo、会话和 PGA 等。

表 2-2 演示了从数据文件读取一个数据块到 DB Cache 的简单过程，这是老白在参考了 Poder 的讲义后所模仿的。

表　2-2

操　　作	闩锁等待	其他等待	CPU时间	说　　明
				读取（10/2512）
get_latch('cache buffer chains'):spin			1	获得闩锁以便于查找数据
搜索buffer chain			5	查找所需数据
db_file_sequential_read等待		5		正常的IO时间
get_latch('cache buffer lru chains'):spin			10	获取闩锁
get_latch('cache buffer lru chains'):sleep	10			获取不到，休眠
get_latch('cache buffer lru chains'):spin			10	继续获取
get_latch('cache buffer lru chains'):sleep	20			再次休眠
get_latch('cache buffer lru chains'):spin			5	获取到闩锁
查找可用DB BUFFER			3	
写入数据			1	
get_latch('cache buffer chains'):spin			2	获取闩锁以便将cache链入
	30	5	37	

2.3　DB Cache 相关的参数闩锁和等待事件

本节将介绍和 DB Cache 相关的数据库参数、等待事件、系统统计信息、闩锁和相关诊断视图等内容。目的是让读者能够对 DB Cache 及相关的数据库组件有一个总体的认识。由于篇幅的关系，本节仅对这些参数和等待事件做简单的介绍。

DB Cache 相关的参数如表 2-3 所示。

表　2-3

参数名称	参数简介	默　认　值
db_cache_size db_keep_cache_size db_recycle_cache_size db_2k_cache_size db_4k_cache_size db_8k_cache_size db_16k_cache_size db_32k_cache_size	从9*i*开始，设置DB Cache大小的参数。db_cache_size是默认池的大小，db_keep_cache_size是keep pool的大小，db_recycle_cache_size是recycle pool的大小。从9*i*开始还支持非标准大小的表空间，针对每种非标准大小的表空间，需要设置相应的db_nk_cache_size参数。本参数的单位为字节	10*g*，11*g*默认值为0（使用自动共享内存管理）
db_block_buffers buffer_pool_keep buffer_pool_recycle	9*i*以前版本的DB Cache设置参数，在Windows 32位环境下使用3G开关的时候，9*i*数据库也需要使用这组参数。这组参数的单位为块	
db_block_size	数据块大小（9*i*以前）或者标准数据块大小（9*i*及以后版本）	8 KB
db_cache_advice	从9*i*开始引入的参数，db_cache建议的开启开关，如果设为ON，可以看到db_cache建议数据	ON
db_block_checking	当数据块变更（UPDATE，INSERT）时，从磁盘上读取或者在INTERCONNECT传输时进行逻辑一致性的校验。该参数启用后会减少数据块内存损坏或者出现逻辑故障的可能性，不过会带来1%~10%的额外系统负载。数据安全性要求高而且负载不大的系统可以使用，一般系统不建议使用。该参数控制的是用户表空间的数据块，对于系统表空间的数据块，Oracle是必须进行块检查的。11*g*的取值范围：OFF，FALSE，LOW，MEDIUM，FULL，TRUE。10*g* R2的取值范围：OFF，LOW，MEDIUM，FULL。10*g* R1、9*i*的取值范围：TRUE，FALSE	OFF/FALSE
db_block_checksum	控制是否对数据块生成和校验checksum值。这个参数仅控制用户表空间的数据块。对于10*g* R2及更高版本，这个参数的值是STRING类型，取值包括OFF（不生成checksum）、TYPICAL（仅在数据读入和写入时生成和校验checksum，可能带来1%~2%的额外负载）以及FULL（在数据变化时生成和校验checksum，可能带来5%左右的额外负载），默认值为TYPICAL。10*g* R2以前的版本，该参数是布尔型的，默认值是TRUE，和TYPICAL类似	10*g* R2及更高版本：TYPICAL。 早期版本：TRUE
_db_block_lru_latches	定义LRU闩锁的数量，一般情况不建议修改	
_db_block_hash_buckets	db_cache的hash链的数量，一般情况不建议修改	
_db_block_hash_latches	db_cache的hash链的闩锁的数量，一般情况不建议修改	
db_writer_processes	db_writer进程的数量	10*g*及更高版本：1或1/cpu_count。9*i*默认值为1

（续）

参数名称	参数简介	默　认　值
db_file_multiblock_read_count	这个参数在10*g*之前需要设置，控制全表扫描或全索引扫描每次读取的数据块的最大值。针对OLAP系统，这个参数设置略高；OLTP系统，一般默认值就可以了。10*g*以后，数据库会根据I/O情况自动调整，不需要人工干预。11*g*即使设置了，系统也不会采用。9*i*和及之前版本，设置该参数时要十分注意，因为SQL执行计划中计算开销与它有关，如果设置较大的值，优化器计算全表扫描的时候成本会降低	9*i*默认值16；10*g*开始采用自动调整，默认是自动调整

DB Cache 相关的等待事件如表 2-4 所示。

表　2-4

等待事件名称	说　　明
buffer busy waits；read by another session	热块冲突产生的等待。从10*g*开始引入了read by another session这个独立的等待事件，是buffer busy waits中的一个特殊情况
free buffer waits	前台进程无法找到可用的buffer，等待free buffer。如果等待时间较长，说明DB Cache可能不足
checkpoint completed	等待checkpoint完成
write complete waits	前台进程等待DBWR完成写入后查找可用buffer，如果等待时间较长，说明DB Writer的性能不足，或者DB Cache太小

DB Cache 相关的部分统计数据如表 2-5 所示。

表　2-5

名　　称	说　　明
consistent changes	数据块提交了undo信息成为CR块的计数。这个值说明了系统中CR块产生的数量。该值越大，越要注意cache buffers chains等闩锁的情况以及热块对系统性能的影响
consistent gets	一致性读的计数。会话发出的对某个数据块进行一致性读的请求。不能将其与consistent changes混淆。一个CR块产生后，可能被多个consistent gets事件调用，因此该值要比前一个值大得多
data blocks consistent reads - undo records applied	从undo中读取数据，形成CR Read。本计数器记录从undo中获取undo记录的数量。如果这个值较大，说明对于某些修改较为频繁的表的查询和其他操作也很频繁，有可能存在热点表和索引
dirty buffers inspected	当某个会话在LRU链的冷端开始查找空闲的数据块时查到一个脏块，这个值就会增加。如果单位时间内该值较大，说明LRU链的冷端存在较多的脏块。出现这种情况有以下几种可能：（1）系统中的脏块数量十分巨大，而且DBWR的写入速度不足，导致无法尽快将这些脏块写入硬盘；（2）部分buffer特别热，并且被更改的频率特别高，从而导致LRU链的尾端存在大量这样的块；（3）本系统是一个以DWL为主的系统，数据块的变更十分频繁。碰到这种情况，可以关注一下DBWR的性能、DB Cache的命中率及cache buffers chains等闩锁的情况
DBWR buffers scanned	当某些触发条件发生时，DBWR会在LRU链的冷端扫描脏块，组成DBWR batch，这个值统计的是DBWR在LRU上扫描的buffer的总数，包括脏块和干净的块。这个值除以DBWR LRU scans就是每次扫描查找的数据块的数量

（续）

名　　称	说　　明
DBWR checkpoint buffers written	检查点时DBWR写入的脏块的数量。如果在单位时间里这个值比较大，说明系统中数据块的变更较为频繁
DBWR free buffers found	DBWR从LRU链中扫描buffer时发现的空闲buffer的数量。该值除以DBWR make free requests就是每次DBWR在收到DBWR make free消息时，扫描LRU链找到的空闲buffer的平均数。这个平均数一般会比较少
DBWR make free requests	DBWR收到的make free消息的数量。如果某个前台进程无法找到空闲的buffer，就会向DBWR发出make free消息。如果单位时间内这个值较高，说明DB Cache可能不足
DBWR summed scan depth	DBWR扫描LRU链查找脏块时，查找的buffer的数量。这个数越大，说明LRU链尾部的脏块数量越少。从Oracle 8*i*开始，LRU链的算法发生了变化，所以如果LRU链尾部的热块比较多，也可能造成这个值较大
DBWR timeout	DBWR idle超过一个特定值，该值就会加1。如果该值较高，说明buffer cache中的数据变化较小，需要写入磁盘的脏块数量极少
DBWR transaction table writes	DBWR写入的回滚端头的数量。该值较高说明有较多的热块正在被写入，而大量用户进程在等待这些块写入完成
DBWR undo block writes	DBWR写入回滚段的数据块数量
buffer deadlock	DB Cache死锁的数量。如果单位时间内该值较高，可能DB Cache存在性能问题，或者存在某些bug
buffer is not pinned count	被访问时已经释放的buffer的数量。只用于Oracle内部调试，并不说明性能问题
buffer is pinned count	被访问时已经被pin住了的buffer的数量。如果单位时间内这个值比较高，说明可能存在热块
exchange deadlocks	当进行两个buffer交换时，发生内部死锁的计数。索引扫描是导致这种交换的唯一因素。如果该值较高，可以检查是否存在十分热的索引（可以通过buffer busy waits分析来定位）
free buffer inspected	从LRU队列的尾部扫描可重用的buffer时跳过的buffer的数量
hot buffers moved to head of LRU	当一个热块到达LRU队列的尾部时，Oracle会自动把这个热块移动到LRU队列的头部，使之能够继续被使用。每发生一次这样的操作，这个计数就加1。值得注意的是，从Oracle 8*i*开始，LRU的算法发生了变化，通过引入TCH计数来确定热块，而不是通过将热块在LRU链上移动来保证热块不被过早地换出。如果热块存在于LRU链的尾部，扫描时发现了热块会主动地跳过，从而保证热块不被过早地重用。
physical reads direct	直接物理读的数量。读时不经过buffer。一般发生这种操作的情况有：排序操作、并行查询操作的从属进程或者预读
physical writes direct	直接写的数量。不经过buffer，直接写入。一般发生这种操作的情况有：直接装载操作，比如CREATE TABLE AS SELECT；并行DWL操作；排序操作中的临时表空间写入；写入没有缓冲的LOB字段
physical writes non checkpoint	非checkpoint引起的物理写。物理写发生的情况包括checkpoint、无足够的空闲buffer可用，或者DBWR超时等。一般情况下，这个值会超过physical writes的一半以上，除非是checkpoint十分频繁的系统。如果该值占physical writes的比重比较少，应该进行分析
pinned buffers inspected	当一个用户进程扫描replacement列表，寻找可重用的buffer时，发现一个冷块被pin了，或者有一个pin请求的等待事件。这种情况很少发生，因为冷块很少会被pin。如果平均每秒该值较大，需要进行分析
remote Instance undo block writes	如果远程实例需要读取某个undo块，需要这个实例先将这个“脏的”undo块回写，该计数器就会增加

（续）

名　　称	说　　明
remote Instance undo header writes	和上一个值类似，只是写入的是undo header
remote Instance undo requests	由于要做CR而从远程实例中请求undo的数量。如果这个值较大，说明RAC中的某些数据块经常在实例间共享，某个实例修改过的数据也在被其他实例使用。这种情况下，需要留意CLUSTER INTEROBJECT的性能
recovery array read time	恢复时产生的I/O消耗的时间
recovery array reads	恢复时产生的I/O的次数
recovery blocks read	恢复时读取的数据块的数量
write clones created in background	如果当前的buffer正在被写入，那么后台进程或者前台进程克隆一个新的buffer，使原来的buffer的写入可以继续进行

DB Cache 相关的闩锁如表 2-6 所示。

表　2-6

名　　称	说　　明
cache buffer handles	保护访问cache的handle的数据结构的闩锁，一般情况下这个闩锁很少出现严重的争用
cache buffers chains	保护hash链的数据结构的闩锁，当buffer扫描的时候会用到该闩锁。该闩锁争用的主要原因是热块或者热链
cache buffers lru chain	保护LRU链数据结构的闩锁。当查找空闲buffer的操作很频繁时，DB Cache容量不足的时候该闩锁争用会比较严重
checkpoint queue latch	保护checkpoint queue数据结构的闩锁
simulator hash latch；simulator lru latch	从9*i*开始，数据库提供了DB Cache建议功能，该建议可以模拟在各种大小的DB Cache下系统产生的物理读的情况。为了实现这种功能，Oracle使用了一个专门的缓冲池，用于模拟计算使用，这两个闩锁就是保护模拟池的数据结构的。当某个buffer操作产生时，会在模拟池中记录这些信息。DB Cache建议的功能会带来一些额外的系统开销，严重时会引起DB Cache相关的性能问题

2.4　DB Cache 优化的一些探讨

在前面三节中我们探讨了一些关于 DB Cache 的算法，探讨算法并不是最终的目的，实际上我们更为关注的是 DB Cache 的一些内部算法能够为优化工作带来什么帮助。理解 DB Cache 的基本原理对于我们优化热块冲突等问题具有十分重要的意义。

2.4.1　DB Cache 和热块冲突

在大家碰到 cache buffers chains 闩锁竞争时，可能都会想到热块争用（HOT BLOCK），其实不尽然。正如我们前面几节所讨论的，cache buffers chains 是串联 DB Cache 的 Hash 链，而 Hash 链是和数据块缓冲的寻址有关的。当某个数据块被装载到 DB Cache 中的时候，系统会根据 DBA

的散列值（DBA 由数据块的文件号和块号组成，长度为 32 字节，其中前 10 个字节为文件号，后面 22 个字节为块号），找到一个 Hash 链，获取要访问这个 Hash 链的 cache buffers chains 子闩锁，然后把 Buffer Head 连接到这个 Hash 链中。如果要查找某个数据块，系统也会首先根据 DBA 的散列值找到这个 Hash 链，然后在链上搜索。因此，如果出现严重的热块冲突，那么有可能会导致 cache buffers chains 闩锁的竞争，但是闩锁竞争并不一定都是由热块冲突所导致的，至少还有两种情况可能导致 cache buffers chains 争用。

其中一种情况是数据库在某个时段出现了大量的数据块扫描操作（比如大量的对大表的全表扫描操作），那么对于 cache buffers chains 闩锁的争用将会十分严重。这种情况下，数据库级的优化能够产生的优化效果十分有限。修改应用，减少这种全表扫描才是解决问题的根本之道。

还有一种情况是系统中存在热链。如果在某个 Hash 链上的多个数据块都比较热，而且存在大量的 CR Block（因为 CR Block 的 DBA 是相同的，因此某个数据块的所有 CR Block 也会在同一个 Hash 链里），那么这个 Hash 链会变得很热，它对应的闩锁就会有严重的竞争。

如果我们发现 STATSPACK 报告中的 buffer nowait 技术指标是 100%，并且从 buffer busy wait 小节中看不到很严重的数据块的热块冲突，那么 cache buffers chains 闩锁竞争很大程度上可能是由于 CR Get 过多或者热链（Hot Chains）引起的，而不是由于热块引起的。比如下面的例子：

```
                          Child        Get                           Spin &
Latch Name                  Num     Requests    Misses    Sleeps     Sleeps 1->4
---------------------- ------- ------------ ----------- ---------- ------------

cache buffers chains        634    7,403,566    202,385     1,311 201094/1271/20/0/0
cache buffers chains       1826    1,317,329        552        45  507/45/0/0/0
cache buffers chains       1863    1,304,674        565        94  472/92/1/0/0
cache buffers chains       1613    1,085,545        315        14  301/14/0/0/0
cache buffers chains       1195    1,047,463      5,948       268  5683/262/3/0/0
cache buffers chains        939    1,040,673        261        14  248/12/1/0/0
cache buffers chains       1314    1,030,316      6,604        40  6567/34/3/0/0
cache buffers chains       1882      895,423      2,636       109  2531/101/4/0
```

上面的数据是从一个 10 级的 STATSPACK 报告中获取到的，这个报告中包含了闩锁的子闩锁的情况分析。在一般的生产环境中，我们很难采集到如此详细的 STATSPACK 报告，但是在某些特殊的诊断情况下，我们可以视情况的不同而采集级别比较高的采样。在这个报告中，我们可以看到：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                    % Total
Event                                              Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
latch free                                         183,537      42,065    35.47
enqueue                                             34,562      37,809    31.88
PX Deq: Txn Recovery Start                           6,877       7,864     6.63
global cache cr request                          1,284,391       6,855     5.78
db file sequential read                          1,098,682       6,576     5.55
          -------------------------------------------------------------
```

在 Top Event 中，latch free 等待十分严重，平均每次等待的时间也比较长：

```
                                                                   Avg
                                                     Total Wait   wait    Waits
Event                                 Waits   Timeouts   Time (s)   (ms)     /txn
---------------------------- ------------ ---------- ---------- ------ --------
latch free                          183,537          0     42,065    229      5.3
```

平均闩锁等待时间为 229 毫秒，这是一个严重的问题。我们来检查一下 cache buffers chains 闩锁，发现其 pct noWait Miss 相当高：

```
                                            Pct    Avg   Wait                 Pct
                                  Get       Get   Slps   Time       NoWait NoWait
Latch                        Requests      Miss  /Miss    (s)     Requests   Miss
------------------------ -------------- ------ ------ ------ ------------ ------
cache buffers chains        144,311,314    0.3    0.1  10735    5,427,525    4.6
```

我们可以看到，634 号子闩锁的争用尤其严重，其 request、miss 和 sleep 的数量均比排名紧随其后的几个子闩锁要高很多。这就是我们所说的热链。

如果碰到了热链，那么解决问题就不是那么简单了。有一个可以尝试的方法，就是调整参数_db_block_hash_buckets，或者略微调大（或者调小）DB Cache 的大小，这样就可以达到解散热链的目的。可以选取一个当前该参数值附近（略大或者略小）的素数，作为_db_block_hash_buckets 值来解开热链。对于 Oracle 9*i* 来说，可以设置_db_block_hash_buckets 为略大于 db buffer 数量的 2 倍的素数。每个数据库版本下，这个参数的调整方法是不同的，因此在调整前一定要认真阅读相关的文档。

实际上，在碰到 cache buffers chains 闩锁竞争的时候，热链存在的情况是很多的，笔者就曾经通过调整_db_block_hash_buckets 解决过一个大型系统的 cache buffers chains 竞争问题。在调整参数前，平均事务响应时间是 3 秒多，CPU 占用率长时间 100%，latch free 是第一位的 EVENT，调整后，CPU 占用率降低到 65%，平均事务响应时间下降为 1 秒左右。

调整前：

```
cache buffers chains      1,528,766,244    0.1    0.4 ######   65,503,162   23.5
cache buffers lru chain       1,667,682    0.6    0.2    782   69,088,014    3.8

cache buffers chains     kcbgtcr: kslbegin excl          0     227,599  414,454
cache buffers chains     kcbrls: kslbegin                0     199,620   20,593
cache buffers chains     kcbgtcr: fast path              0      73,320   77,189

cache buffers lru chain  kcbzgb: multiple sets nowa #######      2,032      384
cache buffers lru chain  kcbzgb: posted for free bu   1,900        241    1,393
cache buffers lru chain  kcbzar: KSLNBEGIN           62,210        177        0
```

优化后：

```
cache buffers chains      1,404,030,764    0.1    0.1    411   62,908,708    0.1
cache buffers lru chain       2,659,978    0.6    0.1     12   66,829,453    1.6

cache buffers chains     kcbgtcr: kslbegin excl          0      39,086   47,161
cache buffers chains     kcbgtcr: fast path              0      28,329   25,597
cache buffers chains     kcbrls: kslbegin                0      27,203   25,948
```

```
cache buffers lru chain  kcbzgb: multiple sets nowa #######      1,215      111
cache buffers lru chain  kcbzar: KSLNBEGIN            31,894        109        0
cache buffers lru chain  kcbzgm                        4,500         37        0
cache buffers lru chain  kcbbiop: lru scan            15,376         36      114
```

以上数据都是在业务量类似的情况下，1 小时内的采样。大家可能觉得学到了一个优化热链的好方法，其实数据库的情况十分复杂，造成 cache buffers chains 闩锁争用的三个原因可能会产生交叉，实际的生产环境远比我们在理论上的分析复杂得多。如果某个数据块的访问量很大，也就是我们常说的很热，而且这个数据块的 CR Block 数量很多，那么调整 BUCKETS 参数的方法就无法起到很好的作用了。老白曾经和多名资深的 DBA 讨论过_db_block_hash_buckets 参数的调整问题，他们都有过相似的经历，想通过调整这个参数来改善 cache buffers chains 闩锁的争用情况，不过无一例外，他们都失败了。也许是老白的运气很好，正好碰到了一个可以通过简单调整参数来解决问题的案例。

如果能够确定问题是由于热块冲突导致的，那么一般情况下，我们可以通过下面的方法来进行优化。

- ❑ 加大表或者索引的 PCTFREE，使每个数据块中存放更少的行。
- ❑ 减小表空间的 Block Size（9*i* 或者更新的版本），使每个数据块中存放更少的行。
- ❑ 使用 Hash 簇表，打散数据存放的位置。
- ❑ 使用 Hash 分区表，使数据分布更为分散。
- ❑ 使用反转键索引（针对索引），减少类似于主键热块这样的冲突。

实际上，如果我们发现热链是由于大量的 CR Get 引起的，那么就需要认真分析到底为什么会产生这么多的 CR Get，这种情况一般是由大量没有优化的 SQL 引起的，因此在必要时优化应用可能效果更佳。

2.4.2　使用 KEEP POOL 能改善 CBC 争用吗

使用保留池（Keep Pool）能否改善 cache buffers chains 闩锁争用？这个问题是多年前 DBA 界广有争论的一个话题。那时候大家对 DB Cache 内部结构的认识还比较初级，都认为 Keep Pool 能够改善 cache buffers chains 的争用。

随着大家对 DB Cache 内部结构的了解，以及对 Hash 链和 LRU 链内部结构的认识，有一些 DBA 提出了一个论断，就是启用 Keep Pool 仅仅改变了 LRU 链的结构，对于 Hash 链并无影响。Keep Pool 启用后，在 LRU 链结构中，多了一个名为 KEEP POOL LRU CHAINS 的链，所有 Keep Pool 的缓冲区都属于一个 DB Cache 中的子池，这个池中的 Buffer Head 都链在不同的链上，其换进换出算法也和 Default Pool 略有不同。不过无论 LRU 链如何变化，都不会影响到 Hash 链的结构。某张表，比如 T_SMALL 表，无论放在 Default Pool，还是 Keep Pool，其 DBA 都不会变化，DBA 的散列值也不会变化，因此 Hash 链也就不会发生变化。根据这个特点，可以得出一个结论，启用 Keep Pool 无法改善 Hash 链的性能。

这个观点似乎无懈可击，也是目前 DBA 界广为接受的观点，不过老白认为此观点仅仅是局部的真理。从 Keep Pool 的启用对 Hash 链的影响上看，Keep Pool 并不能改善 cache buffers chains 争用。但是，Oracle 永远不会是一个判断题，从另外一个角度来看，由于 Keep Pool 的启用，可以将一些访问十分频繁的数据放入其中，如果 Keep Pool 足够大，那么它的命中率就可以接近 100%，因此启用 Keep Pool，可以改善 DB Cache 的命中率，减少物理读。从这个角度来看，Keep Pool 的启用会减少 SERVER 进程对 cache buffers chains 的 pin 操作，从而就能够减少 cache buffers chains 的争用。

从这个例子我们可以看出，Oracle 数据库是一个十分复杂的综合体系，就像我们地球的生态环境一样，一个变更可能会引起连锁反应，不是简单地用公式就可以描述的。因此仅仅从问题的表层去分析是不能客观反映出问题本质的。分析这个问题，不能仅仅从 Hash 链的结构和 cache buffers chains 闩锁本身来考虑，而应该综合考虑 DB Cache 的总体情况，这样才能得到比较客观的答案。

事实上，启用 Keep Pool 能否改善 cache buffers chains 争用，这个问题还真的不好一概而论，如果 cache buffers chains 争用集中在某些热链上，而 Keep Pool 中存储的对象和这些热链没有任何关系，那么启用 Keep Pool 能够达到的优化效果也会十分有限。

2.4.3　如何判断 DB Cache 是否足够

DB Cache 是否够用，这是 DBA 经常探讨的一个话题。十多年前，内存还是十分昂贵的系统资源，CPU 更是贵如黄金。那时，几百 MB 的 DB Cache 就是很奢侈的配置了。如果能够保证系统有 90%以上的 DB Cache 命中率，就相当不错了。Oracle 的官方文档也指出，当 DB Cache 的命中率小于 90%的时候，就需要扩大 DB Cache。在很长的时间里，通过 DB Cache 命中率来判断其是否充足成为一种标准。很多文档都曾指出，DB Cache 的命中率要超过 95％，否则就需要加大 DB Cache，Oracle 公司也将多个 HEALTH CHECK 脚本判断命中率是否合理的指标定为 90％。

那么到底 DB Cache 的命中率是多少才算合适呢？这可能是很多 DBA 都想知道的，不过这个问题恐怕无解。因为在目前的 Oracle 数据库版本下，通过 DB Cache 的命中率来判断其是否足够，是不科学的。应用系统是千差万别的，不同的系统，在不同的情况下，对于 DB Cache 命中率的要求是不同的。我们分析问题的时候，不仅仅要考虑某些命中率的指标，更重要的是要考虑在当前的命中率前提下，系统 I/O 对系统性能的整体影响，以及目前系统的 CPU、内存的使用情况。

以 OWI（Oracle Wait Interface）的观点和时间模型的观点来分析性能问题是目前 Oracle 数据库性能分析的主流方法，也是较为可取的方法。虽然 Oracle 10*g* 引入了 ADDM 分析等新的概念，不过 ADDM 也是基于 OWI 和时间模型的一种分析方法，只是 ADDM 更为智能而已。在使用 OWI 结合时间模型的方法时，首先我们可以通过时间模型，找到问题的关键点，那些产生最多等待，消耗最多系统资源的地方，就是我们下一步优化的重点。然后我们再来看看哪些等待事件是最为

严重的，这些等待事件是否可以优化。

具体应用到分析方面，DB Cache 的命中率指标可以作为我们分析 DB Cache 的起点。通过命中率指标我们可以初步了解 DB Cache 的总体情况，一般来说，在现在内存价格已经十分低廉的情况下，一个 OLTP 系统的 DB Cache 命中率不应该很低。对于绝大多数 OLTP 系统来说，DB Cache 命中率低于 90%是不可接受的，甚至对于一般的企业级 OLTP 系统来说，低于 95%都是难以容忍的。DB Cache 命中率低，意味着更多的物理 IO、更多的闩锁使用和较低的效率。对于一套 RAC 系统，较低的命中率还意味着 DB Cache 的访问会产生更多的实例间通信消息。因此对于 RAC 系统来说，往往需要有更高的 DB Cache 命中率，甚至在有些 RAC 系统中，98%的 DB Cache 命中率都可能太低了。

通过命中率这一指标，只能让我们对 DB Cache 的运作情况有一个初步的了解，而不能作为我们判断 DB Cache 是否需要调整的依据。判断 DB Cache 是否配置合理，要看系统中是不是存在较为严重的 I/O 等待，系统的 IO 是不是过高，如果减少 I/O 的相关等待，系统的总体性能是否能有所提高。这是一个复杂的系统工程，如果系统中的 db file sequential read 等待十分严重，那么是不是能够加大 DB Cache 来减少这方面的等待，从而提高性能呢？目前系统的 DB Cache 命中率达到了 98％，加大 DB Cache 是否能够取得较好的优化效果呢？这些问题我们需要通过 OWI 分析来找到答案。我们可以从等待事件、闩锁等情况，分析 DB Cache 的设置是否存在问题。

如果你已经熟练掌握了 OWI 分析，那么下一步应该学习更高级的分析方法，就是一种从系统整体出发的分析方法。它结合了 OWI、时间模型、系统综合分析，从系统优化的角度来分析 DB Cache 是否需要调整。做数据库优化，实际上是对系统的 CPU/MEM/IO 资源进行优化组合，通过对这三者的调整，发挥系统的最佳性能。这三者之间的关系如下：

- 加大 DB Cache（MEM），可以减少 IO，但是会增加 CPU 的消耗。
- 减少 DB Cache，会增加 I/O，减少 CPU 的消耗。

一名优秀的 DBA 应该能够在调整的时候，充分考虑系统资源的限制，在使任何一种资源都不出现短板的情况下，提高系统的总吞吐能力。如果当前的系统，I/O 响应时间很正常，I/O 性能很好，那么加大 DB Cache，对系统整体性能的提升是微乎其微的，而对于一个 IO 出现了瓶颈的系统，加大 DB Cache 往往可以获得较好的性能改善。另外，对于一个 CPU 已经十分紧张的系统，盲目加大 DB Cache，可能会导致更为严重的性能问题。因此，我们应该对这些因素有一个综合性的评估。

DBA 应该十分清楚自己的优化目的是什么，是提高整个系统的吞吐能力，还是提高某个业务模块的响应速度。在系统资源很充分的时候，提高某个模块的响应速度是没有多少难度的，但是如果在系统资源很紧张的时候，片面地去提高某个模块的性能，可能会带来很糟糕的结果。

很多 DBA 总在寻找一些方法或理论来支持自己的工作。实际上做优化和练武术一样，练习招术只是很初级的阶段，到达一定层次后，无招胜有招。如果你进入了这个境界，就可以对系统做出任何操作，哪怕是一些反常规的，甚至违背最佳实践要求的调整，但前提是你必须确保结果

是好的。

举个例子，一个客户的业务要对一张超过 1GB 的表频繁地进行随机访问，而且随机访问要求的响应时间十分严格。SQL 是最优化的了，但是响应速度仍然达不到要求。考虑到当时 CPU 使用率还较低（不到 50%），内存也有大量的空闲，我建议将这张表和相关索引放入 Keep Pool，做了这个操作后，系统响应时间达到了客户的要求，CPU 的使用率上升了 10 几个百分点。按常理来说，这种操作是不“正确”的，也是不建议使用的，但是在这个场合，此操作是一个十分有效的解决方案。

通过上面的例子，大家可能也感觉到 DB Cache 的分析和优化确实不是一个简单的问题。那么我们如何来分析和优化 DB Cache 呢？刚才已经提到了，可以先从 DB Cache 的命中率入手，了解 DB Cache 的整体情况。然后再观察系统的 I/O 性能指标是否合理，比如，通过 db file sequential read 和 db file scattered read 这两个等待事件，就可以看出数据库 I/O 的性能。这两个指标在通常情况下都应该是几毫秒，不会超过 10 毫秒，如果小于 4 毫秒，那么说明目前系统的 I/O 性能比较良好。接下来，检查一下 DB Cache 相关的闩锁争用情况，特别是 cache buffers lru chains 闩锁的争用情况。如果该闩锁争用较为严重，那么说明 DB Cache 存在不足。下面我们通过 STATSPACK/AWR 报告的 DB Cache 相关章节来进一步分析：

```
Buffer Pool Statistics for DB: BILL  Instance: xxxx   Snaps: 262 -263
-> Standard block size Pools  D: default,  K: keep,  R: recycle
-> Default Pools for other block sizes: 2k, 4k, 8k, 16k, 32k

                                                                  Free    Write  Buffer
     Number of Cache      Buffer    Physical   Physical  Buffer Complete   Busy
P      Buffers Hit %        Gets       Reads     Writes   Waits    Waits   Waits
--- ---------- ----- ----------- ----------- ---------- ------- --------  ------
D      126,592  97.3  77,269,283   2,100,417     92,173       0        0  87,036
K       63,296  93.5     155,457      10,060        752       0        0       0
          -------------------------------------------------------------
```

在这里，要重点关注每个缓冲池的命中率，因为在前面我们看到的是 DB Cache 的总命中率，而在这里可以看到所有 CACHE 的命中率。上面的例子中，系统有 DEFAULT 和 KEEP 两个缓冲池。Keep Pool 的命中率只有 93.5%，这是一个偏低的值，在一个运行较长时间的系统中，Keep Pool 的命中率应该接近 100%，否则就说明 Keep Pool 可能设置不足。除了 Keep Pool 的命中率外，还需要关注 Free Buffer Waits 和 Write Complete Waits 的指标。如果这两个指标不为 0，那么就需要我们重视了。这两个统计指标在前面章节介绍 DB Cache 内部算法的时候都已经介绍过了，它们都是前台进程查找 free buffer 时碰到的等待，不为零说明前台进程出现过等待 Free Buffer 或者等待 DBWR 写脏块结束的情况。这两项统计数据从另外一个方面说明了 DB Cache 存在严重不足。这些指标比命中率更为关键。

随着硬件价格大幅下降，DB Cache 的调整难度也有所下降，只要系统中存在较为充足的 CPU 和内存资源，那么把 DB Cache 设置得偏大一些，是没有多少风险的。因此我们可以尽可能地配置充足的 DB Cache。

不过在一些存在资源瓶颈的系统中，调整 DB Cache 仍然有一定难度。接下来介绍的这个案

例就很有代表性。

一个客户的系统，每天业务高峰期时 CPU 使用率都高达 80%左右，最近一段时间，甚至达到 100%，如果发现得早，尽快杀掉一些进程，才能够确保系统安全；如果发现得晚，可能导致系统 HANG 死。在 CPU 高位运行的时候，一般来说是不建议加大 DB Cache 的。不过经过分析，我们发现在这个系统中有一个查询语句，正常情况下，在 V$SESSION 中看到的这条语句的数量大概有 50 条，出现问题的时候，这条语句的数量往往会超过 100，甚至接近 200。这条语句在正常时和非正常时的 CPU 消耗是十分接近的，而总体执行时间却相差很大。这就说明在该语句执行过程中，产生了更多的等待，这些等待主要集中在 db file sequential read 和一些与 DB Cache 相关的闩锁等待方面。经过进一步分析，我们发现这条语句访问的分区表的最新分区存储在一个新建的磁盘组上，这个磁盘组是 RAID 5 的，使用了较大的磁盘，不过磁盘的数量较少，因此总体 I/O 能力远低于以前的磁盘组。将这个分区迁出目前的磁盘组是较为彻底的解决方法，不过由于这是一个 7×24 的核心系统，不可能给我们留出足够的迁移时间，我们必须从另外的角度来考虑解决方案。这个时候，就需要逆向思维了。系统的 CPU 为什么会突然升高？那是由于磁盘 IO 性能不足，导致这个查询的时间比正常情况慢了 5～10 倍，由于应用服务器端没有对这类查询进行排队，该模块性能变慢，使得更多的查询请求被积压，同时在系统中处理的该查询数量增加，并发访问量增加了数倍，再加上闩锁争用使 CPU 资源消耗量不断提高，最终导致系统过载。分析清楚了这一点，下一步就要考虑，如果我们加大 DB Cache 会怎么样呢？会不会导致 CPU 资源消耗进一步恶化呢？加大 DB Cache 后，I/O 会有所减少，这个 SQL 的 I/O 等待会降低，cache buffers chains 闩锁争用也会有所降低，那么这个 SQL 的执行时间就会缩短。如果 SQL 的执行时间缩短到一定程度，这个查询就不会产生长时间的积压。如果不产生长时间的查询积压，就不会达到 CPU 资源的临界点，那么系统故障也就能避免了。因此我们决定加大 DB Cache，操作完成后，I/O 性能确实有所改善，这个 SQL 的执行时间也缩短了，虽然偶尔也会产生该查询的积压，但是很快就会逐渐减少。

上面的案例通过一个看似会恶化 CPU 资源的操作，居然解决了 CPU 的问题。这是什么道理呢？实际上，这个问题并不是真正的 CPU 资源出现了瓶颈，最终系统崩溃的时候确实是 CPU 资源出现了不足，但是资源不足是由于应用架构不合理导致了大量查询积压，积压到一定程度才导致 CPU 资源不足。在这种情况下解决问题不能拘泥于陈规，只要想办法不让查询积压，那么 CPU 资源不足的现象就不会出现，这个问题也就解决了，所以我们果断地提出了加大 DB Cache 的方案。

熟悉的人总是说我有很多奇思妙想，做事完全不受条条框框的限制。以我这些年做性能优化的经验，操作层面上的内容是很简单的，但是要达到无招胜有招的境界，就必须对 Oracle 的基本原理十分熟悉，要明白每个操作可能产生的后果，并且从中排除主要风险。在调整 DB Cache 的时候也是这样，每个操作之前，必须应用 DB Cache 的基本原理，分析可能会带来的影响。有一次我们为客户做优化，由我们的一名资深工程师实施完成，通过优化，系统响应时间缩短为原来的十分之一，CPU 使用率从 100%下降为 50%左右。后来一个 Oracle 的朋友问我是怎么做到的，我说其实很简单，做表分析，要求客户将 SQL 中的提示全部去掉。因为这个项目 Oracle 也介入过，但

是没有找到很好的解决方案，他感慨道，这些操作很容易，但是敢于完成这个操作的人，才是真正的高手。确实是这样的，真正的高手并不是拥有一些所谓独门秘笈的人，而是能够做出合理判断的人。

2.4.4 DB Cache 优化要点

通过前面章节关于 DB Cache 的一些基本概念的介绍，大家可能已经了解了什么是 DB Cache，以及 DB Cache 是如何工作的。可能有些读者已经能够悟出调整 DB Cache 的一些思路了，不过这方面经验不是很丰富的读者，可能还是存在一定的疑惑。经常有朋友问我，该如何设置 DB Cache 呢？95%的命中率是否说明 DB Cache 是充足的？

实际上 DB Cache 的本质就是一个缓冲区，可以让数据文件中的数据尽可能一次读取多次持用。因此 DB Cache 的优化并无一个明确的命中率指标，而需要根据实际情况来具体分析。对于一个存储性能十分优秀的系统来说，只要存储的能力是充足的，那么哪怕 DB Cache 设置得偏小，命中率偏低，系统的总体性能也还可以忍受，客户也不一定能够发现系统性能存在问题。而对于一个 I/O 性能不佳的系统，DB Cache 的命中率稍微下降一点，可能就会引起十分严重的性能问题，客户端就能够很明确地感受到性能下降。因此，DB Cache 的优化和系统整体资源的情况也是息息相关的，不仅仅取决于数据库的性能指标。

前面的章节已经介绍了很多 DB Cache 的内部原理，可能很多读者在阅读这部分内容时会感觉头昏脑胀，昏昏欲睡。不过这也没关系，因为对于绝大多数 DBA 来说，不需要了解特别多的 DB Cache 内部原理的细节。但是 DB Cache 的性能是对 Oracle 数据库性能影响最大的，因此如何管理好 DB Cache 是十分关键的。我们对于 DB Cache 的理解可以比较粗浅，但是对于本节介绍的 DB Cache 的优化要点，需要认真研究，并且一定要真正地掌握。

对于 DB Cache 的优化，老白根据这些年的经验，总结了以下的几个总体建议。

- 在内存、CPU 资源都充足的情况下，尽量将 DB Cache 配置得大一些，防止在业务峰值时 DB Cache 不足。
- 尽量使用多缓冲，使用 KEEP POOL、NK_POOL 之类的缓冲池，虽然使用多缓冲会加大 DBA 的工作量，但还是值得的。使用了多缓冲，就要考虑哪些对象放入哪些缓冲（NK 缓冲涉及不同 Block Size 的表空间，放入该表空间的对象会被自动放入对应的 NK 缓冲）。
- 无论如何配置，千万别让操作系统产生换页。对于 UNIX 系统，还要注意调整操作系统的虚拟内存参数，比如 AIX 系统的 MAXPERM%参数，避免 FILE BUFFER/CACHE 占用大量的内存。虽然 AIX 可以通过参数调整，在物理内存不足时，尽可能将非计算内存换出，而不会换出计算内存，但是如果系统的物理内存空闲总是十分小（比如说只有几百兆），突然出现几个大查询将物理内存耗尽了，那么操作系统就会产生换页。首先要将部分非计算内存换到交换区，然后释放内存给相关的会话使用。这时，换页会消耗大量的 CPU 资源，从而导致某些业务模块变慢。目前的绝大多数应用系统都没有设计应用访问

队列的机制，无法主动平滑系统峰值。当某个常用查询模块变慢时，就会有更多的查询请求涌入，从而导致会话数量猛增，产生更多的物理内存需求，需要操作系统换出更多的非计算内存，以满足当前的物理内存需求。这样可能导致换页操作持续几分钟甚至几十分钟。在这个时间段内，就可能出现严重的性能问题，对于 RAC 系统，甚至可能出现脑裂的危险，导致某个节点被迫重启。AIX 系统的 MAXPERM%默认值为 80%，HP-UX 默认的文件缓冲可使用物理内存的最大比例是 50%，这些设置，对于数据库服务器来说都太高了（相对来说 SUN 的操作系统比较合理，类似的参数默认值是 12%）。无论是否使用裸设备，操作系统的文件缓冲区对于 Oracle 来说都是没有多大作用的，即便是纯粹的 RDBMS SERVER，MAXPERM%设置为 10 也已经够高了。

- 如果 I/O 存在问题，除了修改 SQL、做负载均衡外，最可能的解决方案是调整 DB Cache，加大 DB Cache 可以减轻 I/O 的负载，提高 I/O 的总体响应速度。
- 加大 DB Cache，可以减轻 I/O 的负载，但同时可能会消耗更多的 CPU 资源。前面提到，绝大多数应用系统都没有设计应用缓冲队列，因此缺乏平缓峰值的功能。如果加大了 DB Cache，减轻了 I/O 的负载，整个系统的处理能力就会得到提升，那么单位时间内处理的事务数量也会增加。原本在应用服务器端排队的访问请求会更快地将数据库操作加载到数据库服务器上，因此数据库服务器的 CPU 消耗可能会有所提升。在 CPU 资源已经十分紧张的情况下，大幅度加大 DB Cache 就是一件要十分慎重的事情。
- 如果出现 I/O 问题，除了看 DB Cache 的命中率外，最应该看的是 STATSPACK 里的 FILE I/O、LATCH 这些节，当然如果有 TOP OBJECT 的话，可以获取更多的辅助信息。对于 LATCH 这节，一定要观察 Latch Miss Sources，这里会体现更多的细节。
- 使用 10*g* 的朋友，不要盲目相信 Oracle 的自动共享内存管理（ASMM），在使用 ASMM 的同时，也要防止共享内存出现严重的抖动。如果共享内存中的 DB Cache 和 SHARED POOL 总是在不停地调整，那么可能会对系统的性能产生更为严重的负面影响。最为显著的特点就是，Cursor 相关的等待事件会大量出现。类似“cursor:pin S wait on X”这样的等待可能会让系统出现严重的性能问题，甚至短暂地挂起。

其实上面的几点，概括起来就是如果物理内存足够，而且 CPU 资源充足，那么尽可能将 DB Cache 设置得大一些，加大 DB Cache 一般不会有太大的副作用。如果你能把整个数据库都放到 DB Cache 中，那么一般来说 DB Cache 也不会出现什么性能问题。而较小的 DB Cache 带来的性能问题往往会比较严重。

对于 CPU 资源十分紧张的系统，就不建议随意加大 DB Cache 了。前几天碰到一个客户，他们的系统刚刚上线不到三个月，就出现了 CPU 资源紧张的现象，每天的业务高峰期 CPU 使用率都达到了 90%左右。后来他们将 SGA_TARGET 调小后，CPU 资源的使用率下降了 10%。虽然说减小了 SGA_TARGET，不过从 AWR 报告上看，DB Cache 的命中率也在 97%～98%之间，仍然处于合理的范围内。db file sequential read 的平均响应时间在 8 毫秒左右，也基本上能够接受。这样的调整，虽然降低了总体的系统性能，但是主要核心模块的响应时间还在合理的范围内。CPU 使用率的下降，降低了突发事件导致 CPU 出现严重瓶颈的机会。这种调整从整体上来说还是成

功的。

如果减小 SGA_TARGET 导致了严重的 SGA RESIZE，或者 cache buffers chains 闩锁的争用十分严重了，那么这个调整就是不合理的，有可能会引起更为严重的性能问题。

因此调整 DB Cache 的时候，特别是减小 DB Cache 的时候，需要 DBA 有充足的分析，否则很容易引起负面的影响。这时，AWR 报告中的 Buffer Pool Advisory 小节的内容就是一个很好的参考。我们可以根据系统的评估，判断减少 DB Cache 后可能带来的 IO 的增加情况，从而预测减少 DB Cache 后系统是否会产生严重的性能问题。

第 3 章

理解共享池

3

共享池是 Oracle 数据库中一个十分重要的缓冲池。实际上我们在 SQL*Plus 中使用 SHOW SGA 命令就可以看到 SGA 的几个主要组成部分：

```
SQL> show sga

Total System Global Area   1048576000 bytes
Fixed Size                    1271444 bytes
Variable Size               511707500 bytes
Database Buffers            532676608 bytes
Redo Buffers                  2920448 bytes
SQL>
```

Oracle 的 SGA 包含固定数据结构部分（Fixed Size），数据块缓冲区（Database Buffers）、Redo Log 缓冲区（Redo Buffers）、共享池（包含在 Variale Size 中）等几大部分。固定数据结构部分包含了数据库的一些固定的数据结构，包括所有其他共享内存结构的地址和指针等。

共享池是 SGA 中的一个组件，传统上我们认为共享池包含字典缓冲（row cache）和库缓冲（library cache）两个部分，实际上共享池的结构要复杂得多，共享池中还包含数据库的一些十分重要的数据结构，比如资源、锁等。通过下面的 SQL，我们可以了解到共享池中包含哪些组件。

```
SELECT pool, name, bytes FROM v$sgastat where pool='shared pool';
```

顾名思义，共享池是为了让大家共享数据而设置的缓冲池。共享池中有很多组件，而且共享池也是很多会话为自己执行 SQL 分配共享内存的缓冲池，这些在共享池中分配的内存在会话之间使用十分频繁，应实现一次分配多次使用，并且能够被实例中的所有会话共享。DB Cache 可以根据数据块大小很规则地进行均匀分配，因此不会出现碎片问题，而且 DB Cache 的分配和归还也相对简单。和 DB Cache 不同，由于各个会话在共享池中分配空间的时候，所需要的空间差异很大，有的只有几十字节，而有的需要几百千字节甚至几兆字节，因此共享池的分配和归还算法十分复杂。为了确保共享池中共享数据的访问性能，共享池的每次内存分配都必须是连续的内存空间。由于这个特点，经过一段时间的运行，共享池中或多或少都会出现一些碎片，这些碎片分布在一些连续分配的空间之间。

共享池中分配的内存空间，有些是永久使用，不会释放的，这些内存空间被标识为 permanent，这些结构大多数是在实例启动时分配的，也有一些是在系统运行过程中临时分配的，不过这些内

存的特点是只分配，在实例关闭前基本上不会释放。最为典型的永久性内存包括进程信息数据、会话信息数据和一些特殊用途的内存段。进程信息和会话信息所使用的内存是根据参数 processes 和 sessions 在实例启动时一次性分配的，一旦分配就不会变动，也不会动态扩展。因此加大 processes 参数，必须重启实例才能够起作用。另外一些特殊用途的内存数组也是根据参数在实例启动时进行分配的，但是在数据库实例运行过程中，如果这些数组出现不足，是可以动态扩展的。这些动态扩展的内存也是 permanent 的，一旦分配就不会释放。这些数组包括 enqueues(lock)、enqueue resource、transactions、transaction branches 等，对于 RAC 系统，还有 gcs resource、ges resource 等。这些动态扩展的内存结构，如果经常发生扩展，将会在共享池的连续内存空间中，产生大量的不可释放的小碎片，从而导致共享池碎片化。

3

一个最为典型的案例就是在 Oracle 9*i* RAC 中经常由于 GES RESOURCE 数据结构的扩展导致共享池碎片问题。GES RESOURCE 资源在数据库启动时从共享池中分配内存，随着 RAC GLOBAL ENQUEUE 的使用量的增长，如果 GES RESOURCE 出现了不足，那么这个数据结构会自动扩展，而且这些扩展的数据是 permanent 的，只分配不释放。在 9*i* RAC 数据库中，每个 GES RESOURCE 分配几十字节，而且每次只扩展一个 GES RESOURCE。随着 GES RESOURCE 不断增长，共享池碎片化趋势就会越来越严重，最终甚至完全碎片化。这时，即使我们通过 ALTER SYSTEM FLUSH SHARED_POOL 来整理共享池，效果也不大。随着共享池碎片化加剧，这个系统最终可能由于 ORA-4031 而宕机。解决这个问题可以从两个方面去考虑，第一个方面是加大 GES RESOURCE 的初始分配值，使之不产生动态分配；第二个方面是在扩展 GES RESOURCE 的时候，每次扩展一组，而不是一个，从而减少扩展的次数。我们可以通过设置足够大的_lm_locks 和_lm_ress 参数来解决第一个问题，这样数据库启动时就会分配足够多的 GES RESOURCE。对于第二个问题，10.2 版已经对 GES RESOURCE 的扩展方法进行了修改，不再是一次扩展一个资源，而是扩展一组。因此，这个故障在 Oracle 10*g* R2 数据库中很少会出现，但不排除在一些极端的情况下仍会出现，这时，就需要考虑设置那两个隐含参数了。

共享池中还有一些分配的内存是可释放的，这些内存被标志为 freeable 或者 recreateable。freeable 的内存是可以直接释放的，而 recreateable 的内存在 unpin 后也是可以释放的。因此，它们都是可以重用的内存。

共享池的内存是通过 Oracle 通用内存管理（generic Oracle memory manager）来进行管理的，这个管理机制也就是我们常说的 KGH heap Manager。在 KGH heap Manager 机制下，所有共享池的 free 内存都被挂在称为 freelists 的空闲链表上，这个空闲列表是按照 bucket 的机制建立的，根据空闲内存片段的大小，挂在不同的 bucket 上。以 Oracle 9*i* 为例，共享池的 freelists 包含 256 个 bucket，每个 bucket 上链接了不同大小的空闲内存块。具体情况如下：

- 小于 812B 的 bucket 是以 4B 为步长增长的，如 16B，20B，24B，……，812B；
- 超过 812B 的 bucket 是以 64B 为步长增长的，如 876B，940B，……，4012B；
- 超过 4012B 的 bucket 是以 4096 的倍数倍增的，如 4108，8204，16396，32780，65548，……

当某个会话需要从共享池中分配空间的时候，会根据自己分配的大小找到某个 bucket，然后找到一个空闲的内存，从中分配。分配产生的剩余内存，会被挂到相应的 bucket 上，供其他会话

分配使用。而被释放的内存又会被挂到 Freelists 上，为了确保共享池的内存运行一段时间后不会碎片化，被释放的内存会自动进行合并，如果释放的两个内存片段是相邻的，Oracle 会将它们自动合并为一个大内存。

共享池的大小通过参数 shared_pool_size 来定义。在 Oracle 共享内存自动管理或者 Oracle 自动内存管理模式下，可以不设置 shared_pool_size 参数，由数据库自动来根据系统负载分配共享池内存。如果这时我们设置了 shared_pool_size 参数，那么这个参数会作为共享池的初始大小和最小值。无论内存如何自动调整，共享池的大小都不能小于这个值。

共享池及其内存管理算法都十分复杂，如何从这些纷繁的算法中掌握最为基本的方法，从而帮助我们分析问题解决问题呢？接下来的这一节中，老白将会和大家一起来理解共享池中的那些著名的算法，并通过对这些知识的理解，学会分析共享池的问题。

3.1　共享池堆的内部结构

要想深入分析共享池内存的管理，必须要知道 Oracle 的内存空间分配是采用堆管理（HEAP）的模式，堆管理的基础就是 KGH。因此本节将会介绍一些稍微深入一点的 Oracle KGH 的基本原理。这部分内容可能有些枯燥，甚至对于某些人来说可能有点难，不过没关系，如果你真的没有兴趣看下去，可以直接跳过本节，这样恐怕你会在理解共享池内部空间管理的一些深入算法时产生一些偏差，而不会导致你无法理解共享池的基本算法。可以说，本节是专门为希望了解 Oracle 内存管理细节的资深人士准备的，如果你暂时无法理解这些细节，就完全可以忽略。另外本节涉及的内容对于没有开发经验的 DBA 来说，可能会有些困难。

Oracle 中最常见的内存堆包括 SGA HEAP、PGA HEAP，另外，我们最常见的表也是 HEAP TABLE，其空间管理的基本概念都是使用 KGH。

Oracle 的各种内存组织都是堆形式的，每个堆包含一个堆句柄和一系列的内存扩展，每个扩展又包含了一系列连续的 Chunk。内存申请者通过在堆上申请空间的模式来获得内存。我们常见的 SGA、PGA 都是以堆的形式管理的。在堆上分配空间（Chunk）时，根据 ALLOCATION CLASS 的不同，其管理模式也不同。Chunk 是一个连续的内存片段，这些内存片段可以分为以下几类。

- PERMANENT：这类 Chunk 是通过 KGHACPERM 标志来分配的，一旦被分配，在整个堆的生命周期里就不能被解锁和释放。一些常用的、容易产生内存碎片的 Chunk 一般采用 PERMANENT 方式分配。
- FREEABLE：这类 Chunk 是通过 KGHACFREE 标志来分配的，一旦使用完毕，就可以立即使用 kghfre()来释放。
- RECREATEABLE：这类 Chunk 是通过 KGHACRECR 标志来分配的，可以被锁住或者解锁。调用者可以使用 KGHPIR 来锁住这个 Chunk，也可以通过 KGHUPR 来解锁这个 Chunk。当这种 Chunk 被解锁后，通过 LRU 机制进行管理。当调用 kghupr()的时候，调用者会传递一个 LATCH 参数给 HEAP MANAGER，HEAP MANAGER 在使用 kghfre()释放这个 Chunk 的时候，会使用此 LATCH。

- FREEABLE WITH MARK：类似于 FREEABLE，不同的是这类 Chunk 包含一个 kghmrk()，当 Chunk 使用的空间达到 kghmrk()对应的值时，该 Chunk 才能被释放。

另外，不管堆类型如何，都可以拥有子堆，比如 SHARED POOL 就是 SGA HEAP 的子堆。

下面我们来看看如何创建堆。堆创建包含两个步骤，一是初始化堆的句柄，二是给堆分配空间。创建堆时，首先通过 kghini()来初始化堆句柄。一个堆句柄包括一组 FREE LIST BUCKETS 和相应的 SIZE 参数。调用 kghini()时，需要使用两个十分重要的宏：KGHDS 和 KGHDSSIZ，前者定义了堆的 FREE LIST BUCKETS 的数量，后者定义了句柄的大小。调用 kghini()的重要参数包括：

- 堆的扩展大小（EXTENT_SIZE）;
- 堆的 FREELIST BUCKETS 的数量（FREE_NUM）;
- 包含每个 FREE LIST BUCKETS 大小的数组（FREE_SIZE）;
- 包含每个 FREELIST BUCKETS 类型的数组（FREE_\TYPE）。

每个 FREE LIST 包含相同的 ALLOCATION CLASS 的 Chunk。当一个 Chunk 释放的时候，会被放入到小于或等于该 Chunk 的 FREE LIST 中去。在 FREE LIST 里，Chunk 不按照大小排序。

通过 kghini()的参数，可以猜测 Oracle 堆的基本思路，比如，我们从 SGA 中分配一个堆用于共享池，在分配共享池的时候，需要确定每个堆的扩展大小。另外还需要知道堆的 FREE LIST BUCKETS 的数量，这一点很多 DBA 在学习共享池内部原理的时候都已经有所了解，对于 Oracle 8*i* 或者更早的版本，共享池的 FREE LIST 包含 10 个 Bucket，从 9*i* 开始，FREE LIST 包含 256 个 Bucket。同时，我们还需要一个数组，来指明 FREE LIST BUCKETS 中每个 Bucket 存放的 Chunk 的大小，同时用一个数组来存放每个 Bucket 的 FREE 类型。通过这些参数，调用 kghini() 就可以从父堆中分配内存了。分配的内存会被挂载到 FREE LIST 上，这时堆就处于可用的初始化状态了。

下面通过一个 Oracle 8.1.5 的共享池的转储信息来验证上述概念。

```
FREE LISTS:
 Bucket 0 size=44
  Chunk  52fb1c4 sz=       60    free
  Chunk  5161394 sz=       40    free
  Chunk  5164904 sz=       48    free
  Chunk  516fbcc sz=       28    free
  Chunk  5195570 sz=       52    free
  ......
 Bucket 1 size=76
  Chunk  5180938 sz=       76    free
  Chunk  51cb58c sz=       84    free
  Chunk  51bf194 sz=       88    free
  Chunk  51af3e0 sz=       84    free
  Chunk  51ac298 sz=       76    free
  Chunk  51c3818 sz=       76    free
  Chunk  51f04d0 sz=       92    free
  Chunk  5212c90 sz=       92    free
  Chunk  529b280 sz=       76    free
  Chunk  52c2248 sz=       84    free
```

```
   Chunk  52c3784 sz=      80    free
   Chunk  52dbb5c sz=      92    free
   Chunk  52df980 sz=      92    free
   Chunk  52e0aa4 sz=      88    free
  Bucket 2 size=140
  Bucket 3 size=268
   Chunk  516325c sz=     472    free
  Bucket 4 size=524
  Bucket 5 size=1036
   Chunk  5160be8 sz=    1080    free
  Bucket 6 size=2060
   Chunk  516d3cc sz=    2812    free
  Bucket 7 size=4108
  Bucket 8 size=8204
  Bucket 9 size=16396
  Bucket 10 size=32780
   Chunk  4c69598 sz=  5204008    free
 Total free space   =  5212456
```

可以看到，共享池的 FREE LIST 共有 10 个 Bucket，其中 Bucket 0 存放所有小于 76B 的 Chunk，Bucket 10 存放所有大于 32780B 的 Chunk。其他 Bucket 存放的都是大于其 Size 大小、小于下一个 Bucket 的 Size 大小的 Chunk。我们要注意的另外一点是，Bucket 中的 Chunk 不是按照大小排序的。

如果需要从 Chunk 中分配空间，可以通过调用 kghalo()来实现，这个调用返回指向 Chunk 空间的指针。在调用 kghalo()时，需要分配的空间的大小通过 REQ_SIZE 参数传递，但是 HEAP MANAGER 不一定分配申请的大小空间，而是根据判断，分配一个被称为 ACTUAL_SIZE 的空间。ACTUAL_SIZE 可能比 REQ_SIZE 大或者小，这一切都是根据当前拥有空间的情况，以及此类 Chunk 的分配特性来判断的，并不是无规律的。从 Chunk 中分配空间的基本顺序如下。

- 首先搜索 FREELIST 上是否存在可以释放的 Chunk。
- 如果没找到，就查找是否有 RECREATEABLE 的 Chunk 可以释放。释放已经使用的 Chunk，需要用到另外一个链表，就是 LRU 链表。LRU 链表根据使用频繁程度，通过一个双向链来串联已经被分配的 Chunk。如果在 LRU 链中找到了可以释放的 Chunk，就通知 HEAP MANAGER 去释放，如果该 Chunk 是被锁住的，首先要解锁，然后才能释放。HEAP MANAGER 在释放内存时，会自动合并碎片。实际上这个释放 RECREATEABLE 的 Chunk 的过程和我们做 FLUSH SHARED_POOL 操作是类似的，只是其规模要小得多。HEAP MANAGER 释放 Chunk 的时候有一个条件，一旦释放后已经存在足够分配的 Chunk，那么这个释放过程就会结束。
- 如果此时还没有找到可以释放的 Chunk，就从父堆里分配空闲的空间。
- 如果父堆无法分配空间，那么就会报错。

在分配空间的时候，如果要在 FREELIST 中查找，那么首先会根据需要分配空间的大小，找到相应的 Bucket（每个 Bucket 都有一个最大值，指明该 Bucket 中的空闲 Chunk 不会超过某个特定的值，找到最大值小于或等于分配值的最大 Bucket），然后搜索整个链表。如果找到了大小正

好相同的 Chunk，那么就直接分配使用。如果找不到合适的 Chunk，就会选择一个稍大的 Chunk，然后分裂空间，将该 Chunk 分裂为两个 Chunk，一个和所分配空间大小一致，一个是剩下的空间。最后将剩下的空间根据大小挂载到相应的 Bucket 的双向链表上。

如果要释放一个扩展的空间，可以通过 kghfre()调用来实现。在调用时，要让所有正在使用这个扩展的会话完成执行，释放或者解锁相关的对象，然后才能释放扩展。

要释放堆中的所有扩展，可以调用 kghfrh()。不过在释放堆之前，必须先释放所有的子堆。在子堆里释放所有的扩展，可以调用 kghfru()。

上面的这些文字，对于没有任何程序开发经验的人来说，确实有些难以理解。其实老白已经将堆的算法做了最大程度的精简，否则大家就更是一头雾水了。DBA 理解堆的一些基本算法有什么好处呢？实际上，我们平时对于 Oracle 的堆管理，特别是共享池的管理，存在一些认识上的误区，而通过对堆的一些基本原理和算法的理解，可以帮助我们纠正这些错误。

首先是共享池碎片化的问题。根据堆的内存管理方法，很容易可以看出，其实碎片化是不可避免的，共享池使用一段时间之后，肯定会出现碎片化的趋势，而且随着数据库实例启动时间的增长，这种碎片化趋势会越来越明显。但是共享池碎片化并不等于共享池就有问题，虽然说碎片化的共享池效率可能会有所下降，但是一般情况下都在可以接受的范围内，因此 DBA 不需要过多地担心这个问题，不要谈虎色变。

接下来要讨论的一个问题是共享池使用率的问题。很多 DBA 看到共享池使用率比较高，就十分紧张，担心共享池快用完了会出问题。如果了解了堆的内存分配和管理策略，就不会有这样的担心了。既然分配了那么多的共享池，那就尽情地用吧，只有把共享池全部用掉了，才能充分达到共享游标和字典的好处。反倒是共享池使用率不高，资源不能充分地利用，才是值得头痛的事情。因此我们不能通过共享池的使用率来判断其是否存在隐患。有时候共享池使用率只有 50%，但是可能性能并不好；而有时候共享池总是接近 100%的使用率，但是却没有任何问题。

第三个要讨论的问题就是，如果共享池的争用很严重，但是共享池的使用率又并不高，这说明什么？这也是困扰了老白很多年的问题。老白也经常碰到一些系统，共享池相关闩锁争用十分严重，大多数情况下，共享池争用严重是由于共享池内存不足导致的，但是进行分析后发现共享池的使用率并不高，只有 60%左右。这是为什么呢？老白了解了共享池内存分配算法后，才想明白了这其中的道理。如果内存分配十分频繁，那么共享池的 FREELIST 上的 Chunk 会被优先使用，共享池的使用率就会持续上升。而如果共享池使用率不高，那么就说明共享池争用并不是由于分配内存十分频繁。那又是怎么回事呢？我想刚刚看了本节的有些朋友已经有答案了，十分频繁的未被分配内存的访问，肯定是访问可重用的数据了。什么情况下才可能出现这样的争用呢？很明显，软分析就是十分典型的这类访问。于是问题就变得简单了，减少软分析就可以缓解这个问题。那么，加大 SESSION_CACHED_CURSORS 吧，老白把这个参数调整到 200，再一分析，共享池的使用率达到了 90%，共享池的争用减少了。

第四个问题是什么情况可能导致共享池出现 ORA-4031 呢？老白最初接触共享池的时候，总觉得共享池既然是可以通过 LRU 算法换出的，那么应该不会出现不足的情况啊。随着对共享池内存分配和管理算法的研究深入，才发现释放共享池内存是有条件的，当前被锁住的内存是不能

释放的。如果当前正好有个会话在执行，那么其游标相关的所有内存对象必须是锁住的，不能随便释放，否则这个 SQL 的执行就会出现问题。正因为这样，在系统并发量很大的时候，由于很多共享池的内存是被锁住无法释放的，共享池的空闲空间就被分割为无数个很小的碎片，可能无法满足一些较大的分配，出现 ORA-4031 也就不可避免了。在这种情况下，我们可以认为共享池不足导致了问题，如果共享池的空间再大一点，出现内存不足的可能性也就会下降。除了共享池不足，还有什么情况可能导致 ORA-4031 呢？前面所说的那种情况，是由于系统并发量很大，大量的共享池内存被锁住而无法释放。一旦系统负载下降，共享池就会逐渐恢复正常。另外一种情况就是，共享池中的一些 LIST 对象（比如 LOCK、RESOURCE 等）初始化分配了一部分空间，如果这部分列表用完了，可以动态扩展。由于这些扩展是永久性的，所以，它们就像共享池中钉下的一颗颗钉子，是无法拔掉的。如果这类扩展十分频繁，那么时间长了，共享池就会被这些“钉子”分割为很多碎片。这种碎片化是永久性的，随着实例启动时间的推移，会越来越严重，而且无法自动或者手工修复。积累到一定程度，这个系统就会出现严重的碎片化问题，从而出现 ORA-4031，甚至导致宕机。对于这种系统，一般来说可以通过三个渠道来优化：第一个渠道是加大共享池，使之碎片化的时间推迟，但是这只是延迟碎片化，不能避免碎片化；第二个渠道是定期重启实例，通过重启实例，恢复共享池的内存空间，确保系统正常运行，不出现宕机现象，不过要做到这一点，对于大多数 7×24 的系统来说十分困难；第三个是扩大经常动态扩展的列表对象的初始大小，减少其动态扩展的次数，最好能够让它们不再动态扩展，这个做法会浪费一定的共享池空间，但却是解决这个问题最好的方法。

似乎问题都解决了，好像共享池也没那么复杂嘛，只要给共享池足够的空间，它就会很好地工作了。实际上并没有这么简单。还是接着刚才 ORA-4031 的问题往下讨论吧。如果分配了很大的共享池，经过一段时间的运行后，它变得很零碎了，这时我们要分配一个较大的连续空间。共享池中没有这么大的 Chunk 可用，那么就会开始释放空间，而释放空间的时候，需要持有共享池的闩锁，因此在释放共享池空间时，为了确保共享池闩锁被持有的时间不会太长，Oracle 内部对释放 Chunk 的数量做了限制，一旦超过这个限制，这个小型的 FLUSH 操作就会停止。这种情况下，虽然共享池中的空闲空间是足够分配的，但也会出现 ORA-4031 错误。另外由于大量碎片释放时，共享池闩锁是被持有的，并发的其他会话也会受到一定的影响。因此在共享池碎片化特别严重的时候，出现共享池性能故障和 ORA-4031 的可能性就会大大增加。所以我们在理解共享池的工作原理并将其应用在实际工作中时，还是不能过于简单化。

3.1.1 进一步了解共享池

顾名思义，共享池就是 Oracle 数据库所有会话所共享的缓冲池。共享池里存放的也是需要给所有会话共享的数据。前面已经简单介绍过一些共享池的内容，以及内存分配释放的算法，这里就不多做描述了。本节主要介绍共享池的基本构造以及相关的算法。通过对这些知识的了解，读者可以掌握一些共享池分析和优化的思路。

一般情况下，学习共享池的知识以及优化的技巧，不一定需要转储共享池。不过对于想更加深入了解共享池的朋友，转储共享池则是学习共享池内部结构过程中必不可少的步骤。老白要事

先声明的是，共享池的结构十分复杂，不建议初学者去分析共享池的转储，如果你对共享池还缺乏深入的了解，甚至不知道共享池的主要用途是什么，那么面对几百兆甚至几十千兆的转储文件，恐怕会崩溃的。想要深入了解共享池，就需要了解把共享池的内容转储到文件的基本命令，以下命令可以把 LIBRARY_CACHE 的内容转储到 TRACE 文件中：

```
ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level <level>';
```

其中<level>代表 Level 级别，对于 9.2.0 及更高版本，不同 Level 的含义如下所示。

- Level=1，转储库缓存统计信息。
- Level=2，转储散列表概要。
- Level=4，转储库缓存对象，只包含基本信息。
- Level=8，转储库缓存对象，包含详细信息（包括 child references、pin waiters 等）。
- Level=16，增加堆大小信息。
- Level=32，增加堆信息。

和前一节所介绍的堆管理一样，共享池也通过 Free List 管理空闲块，Free List 按不同大小划分 Bucket。在 Oracle 8*i* 中，Oracle 把所有的空闲内存链接在 10 个 Bucket 上，在 Oracle 9*i* 中，Oracle 把空闲的内存链接在 255 个 Bucket 上。

Oracle 8*i* 管理 Free List 的方法是，把所有的空闲块都放在 10 个 Bucket 中，小于 76B 的块都位于 Bucket 0 上；大于 32780B 的块，都在 Bucket 10 上。数据库启动以后，共享池中大多数是连续内存块，当空间分配使用以后，内存块开始被分割，碎片开始出现，Bucket 对应的 Chunk 链表开始变长。当数据库中请求分配共享池空间时，首先进入相应的 Bucket 进行查找，如果找不到，则转向下一个更大的内存块 Bucket，获取第一个 Chunk，分割这个 Chunk，剩余部分会进入相应的 Bucket，进一步增加碎片。最终的结果是，Bucket 0 上的内存块会越来越多，且越来越碎小。在早期的 Oracle 版本中，如果每个 Bucket 上的 Chunk 多于 2000 个，就被认为是共享池碎片过多，不过随着共享池的容量越来越大，经过一段时间的运行后，Bucket 0 的 Chunk 数量可能变成一个十分庞大的数字，高达数万甚至更多。而在大多数情况下，我们请求的都是相对较小的 Chunk，这样搜索 Bucket 0 往往消耗了大量的时间以及资源。这可能会使共享池闩被长时间持有，导致更多的共享池竞争。对于 Oracle 9*i* 之前的数据库，由于 Bucket 的数量太少，因此如果共享池比较大，实例启动一定时间后，Bucket 0 的 Chunk 数量会变得十分巨大，共享池的效率会有所下降，闩锁争用也会越来越严重。这个时候，如果刷新一下共享池，系统性能会有很大的回升。实际上，很多 Oracle 9*i* 的系统也存在这样的问题，共享池碎片化的程度会相当高，系统运行一段时间后就必须刷新共享池了，特别是那些会话数量很多、并发量很大的系统。几年前，老白就碰到过这样一个案例，这个系统有上千个会话，业务高峰期活跃的会话数量高达 200～300 个，而数据库服务器的配置很低，仅有 16GB，因此能够给共享池分配的空间也十分有限，这个系统运行一段时间后性能就会明显下降。经过分析，我们发现是共享池碎片化导致的性能下降，于是建议客户每天晚上业务不忙的时候手工刷新一下共享池，系统就变得比较稳定了。

在 Oracle 9*i* 中，Free Lists 被划分为 0～254 共 255 个 Bucket。每个 Bucket 容纳的大小范围

如下。

- ❑ Bucket 0 ~ 199 容纳的大小以 4 递增。
- ❑ Bucket 200 ~ 249 容纳的大小以 64 递增。
- ❑ Bucket 249：4012 ~ 4107=96。
- ❑ Bucket 250：4108 ~ 8203=4096。
- ❑ Bucket 251：8204 ~ 16395=8192。
- ❑ Bucket 252：16396 ~ 32779=16384。
- ❑ Bucket 253：32780 ~ 65547=32768。
- ❑ Bucket 254：≥65548。

在 Oracle 9*i* 中，对于较小的 Chunk，Oracle 增加了更多的 Bucket 来管理。0 ~ 199 共 200 个 Bucket，大小以 4 为步长递增；200 ~ 249 共 50 个 Bucket，大小以 64 递增。这样每个 Bucket 中容纳的 Chunk 数量大大减少，查找的效率得以提高。这就是 Oracle 9*i* 中共享池管理的增强，通过这个算法的改进，Oracle 8*i* 中过大共享池带来的闩锁争用等性能问题在某种程度上得以解决。

通过内部视图 X$KSMSP 可以监控共享池碎片的情况，这个视图中的每一行代表共享池中的每个块（chunk）。以下几个字段可以帮助我们了解共享池的情况：

- ❑ KSMCHCOM 是注释字段，每个内存块被分配以后，注释会添加在该字段中。
- ❑ KSMCHSIZ 代表块大小。
- ❑ X$KSMSP.KSMCHCLS 列代表类型，主要有 4 类：
  - ■ FREE：free chunks——不包含任何对象的块，可以不受限制地被分配。
  - ■ RECR：recreatable chunks——包含可以被临时移出内存的对象，在需要的时候，这个对象可以被重新创建。例如，许多存储共享 SQL 代码的内存都是可以重建的。
  - ■ FREEABL：freeable chunks——包含会话周期或调用的对象，随后可以被释放。这部分内存有时可以全部或部分提前释放。但需要注意的是，由于某些对象是中间过程产生的，这些对象不能临时被移出内存（因为不可重建）。
  - ■ PERM：permanent memory chunks——包含永久对象，通常不能独立释放。

通过查询 X$KSMSP 视图来考察共享池中存在的内存碎片数量。这里要注意，Oracle 的某些版本（如 10.1.0.2）在某些平台上（如 HP-UX、PA-RISC、64-bit）查询该视图可能导致过度的 CPU 耗用，这是由 Bug 引起的。（对这张视图的访问会产生 KGHDUP 操作，执行过程中会影响共享池的闩锁，因此在业务高峰期，应尽可能减少对该视图的访问，而且最好由 DBA 手动完成。有很多监控系统经常定期采集该视图的数据，这是存在一定风险的，有时候会导致共享池严重冲突，甚至可能导致实例被挂起。）对应示例如下。

首先准备一个数据库，在刚刚启动的时候，查询 X$KSMSP。

```
SQL> select count(*) from x$ksmsp;
  COUNT(*)
----------
      2691
SQL> SELECT a.ksmchcom, SUM (a.CHUNK) CHUNK, SUM (a.recr) recr,  SUM (a.freeabl)
```

```
freeabl,
  2       SUM (a.SUM) SUM
  3  FROM (SELECT   ksmchcom, COUNT (ksmchcom) CHUNK,
  4          DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
  5          DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
  6          SUM (ksmchsiz) SUM
  7        FROM x$ksmsp GROUP BY ksmchcom, ksmchcls) a
  8            GROUP BY a.ksmchcom;

KSMCHCOM              CHUNK       RECR    FREEABL        SUM
---------------- ---------- ---------- ---------- ----------
Global Context            1                   224        224
KGK contexts              2                  2376       2376
KGK heap                  2       3812                  3812
KGL handles             554     186404                186404
KGLS heap               446     220536     391028     611564
KQR PO                  653     296324                296324
KWQI bufq Heap            1                   200        200
LISTEN ADDRESS            9                  1676       1676
LISTEN ENDPOINT           1                  1032       1032
MS alert log              1                 10252      10252
MTTR advisory             4        276      12144      12420
PARAMETER ENTRY           3                    92         92
PARAMETER TABLE           1                  1032       1032
PLS cca hp desc           1                   200        200
PLS non-lib hp            1       2096                  2096
PRESENTATION EN           1                    20         20
PRESENTATION TA           1                  1032       1032
SERVICE NAME EN           2                    76         76
SERVICE NAMES T           1                  1032       1032
character set m           7                142040     142040
character set o           5                275892     275892
fixed allocatio          47       1504                  1504
free memory             140                         67194772
joxs heap                 1                    92         92
joxs heap init            1       4248                  4248
library cache          1696     248812     432356     681168
listener addres           1                    16         16
multiblock rea            2                  4144       4144
obj htab chunk           12                 86208      86208
permanent memor           2                         29302084
reserved stoppe          12                              240
service names a           2                    36         36
session param v           9                152712     152712
sim memory hea           20       4248      80712      84960
sql area                505     489056    1109156    1598212
table definiti            3        252                   252
trigger defini            2        324       1072       1396
trigger inform            2        336        816       1152
trigger source            1         88                    88
已选择 39 行。
SQL>
```

我们记下空闲内存的情况，包括其块的数量，接下来先随便执行一个 SQL 语句，比如：

```
SQL>SELECT *  FROM DBA_TABLES WHERE ROWUNM<10;
```

然后再次执行上面的查询语句查看共享池的情况。

```
SQL> SELECT a.ksmchcom, SUM (a.CHUNK) CHUNK, SUM (a.recr) recr,  SUM (a.freea
 freeabl,
  2       SUM (a.SUM) SUM
  3  FROM (SELECT   ksmchcom, COUNT (ksmchcom) CHUNK,
  4           DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
  5           DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
  6           SUM (ksmchsiz) SUM
  7         FROM x$ksmsp GROUP BY ksmchcom, ksmchcls) a
  8             GROUP BY a.ksmchcom;

KSMCHCOM              CHUNK       RECR    FREEABL        SUM
---------------- ---------- ---------- ---------- ----------
Global Context            1                   224        224
KGK contexts              2                  2376       2376
KGK heap                  2       3812                  3812
KGL handles             582     195836                195836
KGLS heap               450     224376     394564     618940
KQR PO                  703     323068                323068
KWQI bufq Heap            1                   200        200
LISTEN ADDRESS            9                  1676       1676
LISTEN ENDPOINT           1                  1032       1032
MS alert log              1                 10252      10252
MTTR advisory             4        276      12144      12420

KSMCHCOM              CHUNK       RECR    FREEABL        SUM
---------------- ---------- ---------- ---------- ----------
PARAMETER ENTRY           3                    92         92
PARAMETER TABLE           1                  1032       1032
PLS cca hp desc           1                   200        200
PLS non-lib hp            1       2096                  2096
PRESENTATION EN           1                    20         20
PRESENTATION TA           1                  1032       1032
SERVICE NAME EN           2                    76         76
SERVICE NAMES T           1                  1032       1032
character set m           7                142040     142040
character set o           5                275892     275892
fixed allocatio          47       1504                  1504

KSMCHCOM              CHUNK       RECR    FREEABL        SUM
---------------- ---------- ---------- ---------- ----------
free memory             148                         66824732
joxs heap                 1                    92         92
joxs heap init            1       4248                  4248
library cache          1801     263408     462132     725540
listener addres           1                    16         16
modification h            1                  2060       2060
multiblock rea            2                  4144       4144
obj htab chunk           12                 86208      86208
permanent memor           2                         29302084
reserved stoppe          12                              240
```

```
service names a           2                      36         36

KSMCHCOM              CHUNK       RECR    FREEABL        SUM
---------------- ---------- ---------- ---------- ----------
session param v           9                  152712     152712
sim memory hea           20       4248      80712      84960
sql area                584     541728    1336540    1878268
table definiti            3        252                   252
trigger defini            2        324       1072       1396
trigger inform            2        336        816       1152
trigger source            1         88                    88

已选择 40 行。
```


从上面的例子可以看出，空闲内存的容量减少了 370040B，不过它的 Chunk 数量却增加了。这是因为 SQL 解析从空闲内存中分配了空间，并且产生了更多的碎片。从上面对共享池内部管理的算法特点分析，可以得出以下几点结论。

- 比较大的共享池会带来一定的系统开销，因此共享池不是越大越好，特别是在 Oracle 8*i* 或者更早期版本的数据库中。
- 在 9*i* 以后的系统中，特别是 CPU 数量较多的大型 SMP 系统中，较大的共享池带来的负面影响已经大大降低了。大家可以根据自身需要使用较大的共享池。
- 随着系统的运行，共享池的碎片将增加，共享池闩锁竞争也会增加，因此应将一些常用对象固定到共享池中，并且定期刷新共享池，这样有助于保证系统的性能。
- 对于 7×24 的系统，定期刷新共享池有助于保持共享池性能的稳定。

这里我们仅仅从十分简单的角度去分析了共享池空闲空间管理的算法，实际上，Oracle 共享池的管理算法要复杂得多。数据库实例启动并运行了很长时间后，共享池碎片化会变得十分严重，这样就会导致比较大的共享内存分配（比如分配 10KB 的空间）无法满足需求，对于这样的情况，实际上 Oracle 已经有很好的应对措施。Oracle 在共享池中设计了保留池，这个保留池的管理和共享池是不同的。

Oracle 是从 7.1.5 版本开始支持保留池的，保留池的设计目的是为了在共享池碎片化很严重的时候，还能够有一部分保留空间，用于较大的内存分配。我们可以通过参数 SHARED_POOL_RESERVED_SIZE 来设置保留池的大小，不过一般情况下，可以不设置该参数，这样 Oracle 会自动设置保留池的大小为共享池的 5%。但在共享内存不是很大的情况下，5%的空间可能不是十分充足，因此在一些并发量很大的系统中，如果经常出现较大的共享内存分配需求无法满足的现象，那么可以将该参数设置为共享内存的 10%左右。

保留池的使用是有条件的。首先必须是在共享池的 FREELIST 中找不到足够大的 Chunk，其次是分配的内存容量要大于 SHARED_POOL_RESERVED_MIN_ALLOC 的设定值。这个参数从 8*i* 开始已变为隐含参数，即_SHARED_POOL_RESERVED_MIN_ALLOC，其默认值为 4400，也就是说，只有超过 4400B 的内存分配才被认为是大内存分配，可以使用保留池的空间。在一个碎片化很严重的系统中，一旦某个后台进程在执行一个关键性操作的时候无法分配到空间，就有可

能导致数据库实例的崩溃，而一些常用系统对象的大小往往在 4000 ~ 4400B。因此在碎片化很严重且经常由于 ORA-4031 导致宕机的系统中，将这个参数设置得小一点（比如 4000 或者 3800）是十分有必要的。

从 Oracle 9*i* 开始，共享池是可以动态调整的，在数据库不重启的前提下，我们可以通过 alert system set shared_pool_size=...命令来随时动态调整共享池的大小。只要保证调整完毕后总的 SGA 容量小于 SGA_MAX_SIZE 即可。这个新特性给予我们另一个解决共享池问题的思路，在必要时可以加大该参数，使共享池拥有更多的可用内存。如果要享受这个功能的好处，那么我们必须设置 SGA_MAX_SIZE 参数，使之比目前分配的 SGA 的内存总量大一些，这些预留的空间今后将被用于动态扩充某个可动态扩充的缓冲池（比如共享池或者 DB Cache）。实际上 SGA_MAX_SIZE 预留的部分内存并没有被占用，而只是在操作系统中"预约"而已，这部分内存只有在扩充 SGA 中的缓冲池时才会被真正占用。不过，我们在碰到共享池不足的时候，遇到的情况往往是 SGA_MAX_SIZE 参数并未预先设置，这时就只能先减少其他缓冲池的配置，然后再去扩大共享池。这个被迫缩小的缓冲池通常就是 DB Cache。

反过来，在其他缓冲池不足的时候，也可以先动态缩小共享池，然后再去扩充其他缓冲池。不过这种反操作具有一定的风险，对于碎片化很严重并且并发量很大的系统，可能导致系统出现短暂的挂起现象，严重时甚至可能导致实例崩溃。

从 Oracle 10*g* 开始，我们就可以使用共享内存自动管理（ASMM）功能了。这个功能开启之后（通过设置 SGA_TARGET 参数来开启此功能），Oracle 将自动管理共享内存。当共享池不足的时候，Oralce 会自动从其他缓冲池中释放空间，供共享池扩充使用。同样，当其他缓冲池不足的时候，也会自动缩小共享池来扩充该缓冲池。因此，在 Oracle 10*g* 的系统中，我们很少会看到 ORA-4031 这个很著名的错误信息了。Oracle 也在其官方文档中建议 Oracle 10*g* 的用户不要再去设置 DB_CACHE_SIZE、SHARED_POOL_SIZE 之类的参数了，只要设置一个合理的 SGA_TARGET 参数就足够了。

这一切看上去似乎很美，我们只要使用共享内存自动管理就可以了，而不需要再去考虑共享池碎片、ORA-4031。不过现实是残酷的，Oracle 不可能了解每个系统的复杂性。简单地设置 SGA_TARGET 就解决所有问题的想法也有点太天真了。在一个负载较高、相对复杂的 OLTP 系统中，负载的变化是十分多样的。比如，早上 9 点上班的时候，可能突发大量的并发操作，做一些签到、登录的 OLTP 操作，虽然每个操作开销不大，但是并发量很大。这时，DB Cache 压力不大，而共享池压力很大。而由于前一天半夜有很多大型批处理，会使用大量的 DB Cache，因此早上 8 点多钟的时候，共享池已经被压缩得很小了。这时，就需要缩小 DB Cache，扩大共享池。如果恰巧某些部门需要统计一些数据，做了一些大数据量的统计操作，这个 SGA RESIZE 工作就会变得十分复杂，Oracle 的 ASMM 机制可能会无所适从，出现了在短时间内反复压缩扩展共享池的现象，就可能会产生严重的 SGA 性能问题。在这种情况下，我们可能看到大量的游标相关等待，或者库缓存等待。

上述情况，称为 SGA 抖动。为了防止 SGA 抖动，我们可以关闭自动共享内存管理（设置 SGA_TARGET 为 0），也可以为共享池设置一个最小值（通过设置 SHARED_POOL_SIZE 可以设

定共享池的最小值，因为自动共享内存管理缩小共享池时，不能小于该参数指定的值），从而减少共享池抖动，保障共享池的性能不会受到严重的影响。

另外一个关于共享池的热门话题是共享池碎片和共享池碎片化的问题。很多 DBA 的优化目标是减少共享池碎片甚至消除共享池碎片，这其实是一个不可能完成的任务。共享池碎片化的趋势是共享池算法决定的，经过一段时间的运行，共享池的碎片化趋势肯定是越来越明显，碎片也必然越来越多。有些 DBA 通过共享池中平均 Chunk 的大小来判断共享池的性能是否良好，实际上这是一个误区。共享池碎片化的程度越高并不一定性能就越差，很多系统的共享池中 Chunk 都很小，但是共享池的性能并不存在问题。Oracle 一直在想办法设计一个指标，用于评估共享池的性能，不过到目前为止的版本中，还没有提出一个十分具有代表性的指标。根据这些年对共享池的理解，老白觉得每次共享池请求扫描的 Chunk 数量越高，共享池请求的效率就越低，也就说明共享池的碎片化程度越高。不过到目前为止，Oracle 并未提供这个指标。

实际上我们并不需要对共享池碎片化程度进行直接的分析，通过共享池相关闩锁的平均等待时间以及超时的次数，就可以很准确地判断出共享池的效率。

3.1.2　共享池的子池技术

共享池的子池（Sub Pool）技术是从 Oracle 9*i* 才开始引入的。子池技术解决了较大的共享池在并发访问方面的性能问题。在 Oracle 8*i* 和以前的版本中，共享池是没有子池的，整个共享池的分配和释放都由一个共享池闩锁进行串行化控制。

在这种结构下，共享池的分配和释放可能会成为一个瓶颈，当大并发量的共享池分配释放请求存在的时候，共享池闩锁这个瓶颈点就十分明显了，因此引入子池的概念是十分必要的。Oracle 共享池的子池充分利用了大型 SMP 系统多 CPU、海量内存的特点，根据 CPU 的数量来确定子池的数量。其基本原则是，每 4 个 CPU 分配一个子池，最多可分配 7 个子池。那么这里就又有一个问题了，如果某个系统的 CPU 数量很多，而共享池的容量比较小，比如一个包含 32 个 CPU 的系统，分配了 512MB 的共享池，那么按照刚才的算法，共享池可以分为 8 个子池，每个子池的容量就只有 64MB，这么小的子池，很快就会碎片化了。

这一点 Oracle 其实已经考虑到了，无论分配多少个子池，都必须确保每个子池的容量不小于一个最小值，这个最小值在 Oracle 的各个版本中也有所不同。

- 9*i*：128MB。
- 10*g*：256MB。
- 11*g*：512MB。

从上面的数据可以看出，随着 Oracle 数据库版本的更新，子池容量的最小值也在成倍地增加。这说明了两个方面的问题：一方面是随着硬件的发展，共享池的容量也在不断增长；另一方面，Oracle 也已经注意到了，子池过小所导致的共享池碎片问题可能会带来严重的性能问题，因此逐步加大了子池的最小容量。

我们都知道，共享池子池的数量可以通过参数_kghdsidx_count 来指定，其中 kghds 就是 HEAP DESCRIPTION 的结构，如代码清单 3-1 所示。

代码清单 3-1

```
struct kghds {
  void               *指向parent heap 的指针 ;
  ub4                预先分配的大小;
  void *             unpin HEAP 时传入的OWNER 的指针;
  ... ...
  kghhd *            开始分配permentCHUNK 的HD 指针;
  kghlu *            旧堆的指针;
  kghhd *            堆的顶部指针;
  ... ...
  b1                 状态位;
   KGHDSFEMPT        BIT 0x01 //是否包含空闲的EXTENT
   KGHDSFINIT        BIT 0x02 //是否已经初始化
   KGHDSFALGN        BIT 0x04 //是否扩展做了 OS 页面边界对齐
   KGHDSFPRFR        BIT 0x08 //是否永久性的Chunk 分配为 freeable
  ... ...
  ub1                HEAP 中的FREELISTS 的数量;
  ... ...
  char               HEAP 用途描述[15+1];
  b2                 HEAP 的分配类型;
  b2                 HEAP 类型;
  ub2                HEAP 统计类型;
  ub2                HEAP 统计;
  union kghds.UNK_10467782 HEAP 类型;
   KGHDSALOF       BIT 0x01 //总是分配freeable 的CHUNK
   KGHDSFESZ       BIT 0x02 //固定的EXTENT 大小
   KGHDSSESZ       BIT 0x04 //标准EXTENT 大小
   KGHDSFHEAP      BIT 0x08 //不扩展HEAP

}
```

从上面的数据结构来看,在内存结构上,共享池的子池是Shared Pool这个HEAP的Sub Heap。每个子池都有一个 KGHDS结构，要访问这个子池就需要通过 KGHDS结构。Oracle的思路是通过多 CPU 的特性来将一个很大的共享池划分为若干个对等功能的区间，每个独立的区间都有相同的管理机制，可以并发访问，从而解决性能方面的瓶颈。

在具有子池的情况下，Oracle在共享池中分配空间的算法也会发生一些改变。比如，在没有子池的情况下，如果想在共享池中分配空间，必须先闩住共享池闩锁，如果无法获得该闩锁，就需要等待。而在有子池存在的情况下，算法发生了改变，如果某个系统的共享池有6个子池，那么申请共享池闩锁可以从0号子池开始，一直到5号子池，每个子池都有一个共享池子闩锁。在申请0～4号共享池子闩锁的时候，采用不需等待模式，一旦申请失败，不需要等待，直接返回；而申请5号子池的共享池子闩锁的时候，就需要采用等待模式，直到获取这个闩锁为止。

另外一种情况，在分配共享池空间的时候，如果没有子池存在，当共享池没有足够大的块可以使用时，会通过释放一些可释放的块来解决这个问题。而在存在子池的情况下，如果前面的几个子池没有足够大的块可分配，那么它会执行类似无子池的操作呢，还是会继续在其他子池中查找空间呢？由于无法找到关于此算法的描述，因此我们也只能根据以往的工作经验进行推测。代码清单3-2所示的查询结果可能能够解答我们的问题。

代码清单 3-2

```
SQL> column indx heading "indx|indx num"
SQL> column kghlurcr heading "RECURRENT|CHUNKS"
SQL> column kghlutrn heading "TRANSIENT|CHUNKS"
SQL> column kghlufsh heading "FLUSHED|CHUNKS"
SQL> column kghluops heading "PINS AND|RELEASES"
SQL> column kghlunfu heading "ORA-4031|ERRORS"
SQL> column kghluNFS heading "LAST ERROR|SIZE"
SQL> select
  2    indx,
  3    kghlurcr,
  4    kghlutrn,
  5    kghlufsh,
  6    kghluops,
  7    kghlunfu,
  8    kghluNFS
  9  from
 10    sys.x$kghlu
 11  where
 12    inst_id = userenv('Instance');

      indx  RECURRENT  TRANSIENT    FLUSHED   PINS AND   ORA-4031 LAST ERROR
  indx num     CHUNKS     CHUNKS     CHUNKS   RELEASES     ERRORS       SIZE
---------- ---------- ---------- ---------- ---------- ---------- ----------
         0     109782     817050   39672920 1.3463E+10          0          0
         1     226597     953890   41157199 8645642258       2854       3896
         2      34909     570914   41305725 1.0634E+10          0          0
         3      46802     785255   40361546 9380330002          0          0
         4      63663     766774   40506397 8444364769          0          0
         5      39275     762341   40270514 1.0518E+10          0          0
         6     101382     754076   40701740 1.1389E+10          0          0
```

上面的系统中，共享池有 7 个子池，而只有 indx 为 1 的子池出现了 ORA-4031 错误，其他子池中并没有出现。假设块分配算法在某个子池无法分配空间时，就从其他子池中去分配，那么它选择的子池将是无规律的，只根据获取共享池子闩锁的情况来选择，也就是说，ORA-4031 错误不会集中在一个子池中，而会分布在多个子池中；但如果按照 INDX 的顺序有规律地选择子池，那么 ORA-4031 就不应该集中在 INDX 为 1 的子池中，而是应该集中在 INDX 为 6 或者 INDX 为 0 的子池中。因此我们有理由推测块分配的算法是：一旦选择在某个子池分配空间（前提是获取了相关的共享池闩锁），那么当这个子池中无足够的空间可用，而且无法使用保留池时，此次分配就会失败，并出现 ORA-4031 报警。

关于子池更为详细的算法已经不在本节的讨论范围之内了，实际上 DBA 们也不需要那么深入地了解子池的情况。我们只需要从 Oracle 子池的一些基本设计思想和算法中了解共享池子池管理时应该注意的问题。首先必须明确，子池设置的目的是为了提高共享池分配回收和管理的性能，加大共享池的并发访问能力，因此它的存在肯定能够提升共享池的性能。其次，我们必须了解子池技术可能带来的负面影响，由于一个大的共享池会被分割为若干个较小的子池，每个子池独立管理 Free List，因此采用子池后，每个子池的容量变小了，也就增加了共享池碎片化的可能性。

知道了子池技术的两面性，DBA 应该了解在实际工作中如何使用子池技术。在一般情况下，我们可以不用理会它，由 Oracle 自动设置子池即可。而如果共享池经常出现一些碎片问题，甚至是 ORA-4031 错误，那么就必须减少子池的数量，甚至禁用子池。

可能还是有朋友会纠结，究竟使用多少个子池更为合适呢？这一点确实没有定论，系统分配的子池数量在大多数情况下不会存在严重的问题。不过在共享池不是很大而 CPU 数量很多的情况下，我们还是要十分注意的，特别是在 9*i* 版本的数据库中，子池的最小容量为 128MB，这个值一般来说是偏小的，因此从 Oracle 10*g* 开始，子池的最小容量被改为 256MB，这样就更为合理了，而在 11*g* 版本中，子池的最小值又扩大了一倍，这说明 Oracle 也已经认识到过小的子池可能导致的一些负面影响。因此当共享池由于子池而出现了问题的时候，我们可以考虑适当减小 _kghdsidx_count 参数的值，使每个子池的容量大于 256MB，甚至 512MB。当数据库经常出现碎片而导致 ORA-4031 错误的时候，很多技术文档，包括 Metalink 的文章都建议设置_kghdsidx_count 为 1，即关闭子池功能。事实上，这种建议可能会矫枉过正，老白的建议是适当减少，而不彻底关闭。如果真的要关闭子池功能，那么最好能够做好测试，因为关闭子池可能会带来另外一个问题，那就是共享池分配和回收又变成串行化了，这种情况下，共享池闩锁可能会成为瓶颈，因此这种调整可能会带来一些新的问题。

3.1.3　字典缓存

从 Oracle 堆管理（KGH，HEAP MANAGEMENT）的角度来看，字典缓存（Row Cache）是共享池的一个子堆。字典缓存到底存放了什么信息呢？通过下面这个简单的 SQL，我们来看看字典缓存里到底有些什么内容。

```
SQL> SELECT PARAMETER,SUM(COUNT) FROM V$ROWCACHE  GROUP BY PARAMETER ORDER BY
SUM(COUNT);
extensible security principal na          0
realm auth                                0
qmemod_cache_entries                      0
qmrc_cache_entries                        0
qmtmrctq_cache_entries                    0
qmtmrciq_cache_entries                    0
qmtmrctn_cache_entries                    0
AV row cache 3                            0
extensible security principal pa          0
extensible security user and rol          0
Rule Set Cache                            0
dc_outlines                               0
kqlsubheap_object                         0
AV row cache 1                            0
extensible security midtier cach          0
qmtmrcip_cache_entries                    0
qmtmrcin_cache_entries                    0
extensible security principal ne          0
rule_fast_operators                       0
rule_info                                 0
dc_qmc_ldap_cache_entries                 0
qmc_app_cache_entries                     0
```

```
dc_free_extents                           0
dc_tablespace_quotas                      0
realm cache                               0
XS security class privilege               0
rule_or_piece                             0
SMO rowcache                              0
Realm Subordinate Cache                   0
dc_table_scns                             0
Command rule cache                        0
extensible security UID to princ          0
AV row cache 2                            0
qmtmrctp_cache_entries                    0
dc_partition_scns                         0
dc_used_extents                           0
dc_sql_prs_errors                         0
Realm Object cache                        0
dc_awr_control                            1
global database name                      1
dc_profiles                               1
dc_constraints                            1
outstanding_alerts                        5
dc_files                                  6
sch_lj_objs                               7
dc_sequences                              7
dc_tablespaces                            8
dc_object_grants                         12
sch_lj_oids                              16
dc_rollback_segments                     22
dc_global_oids                           30
dc_users                                148
dc_segments                             912
dc_objects                             1625
dc_histogram_defs                      3329
dc_histogram_data                      4414
```

上面的结果是在 Oracle 11.2 数据库中查询得到的，在 9*i* 版本的数据库中结果要简单得多，下面的结果来自于 Oracle 9.2.0.8 数据库。

```
PARAMETER                        SUM(COUNT)
-------------------------------- ----------
dc_app_role                               0
dc_constraints                            0
dc_database_links                         0
dc_encrypted_objects                      0
dc_encryption_profiles                    0
dc_files                                  0
dc_free_extents                           0
dc_outlines                               0
dc_partition_scns                         0
dc_used_extents                           0
dc_tablespace_quotas                      0
dc_table_scns                             0
dc_qmc_ldap_cache_entries                 0
dc_qmc_cache_entries                      0
dc_profiles                               1
dc_sequences                              1
```

```
dc_tablespaces                          3
dc_usernames                            4
dc_user_grants                          5
dc_global_oids                          6
dc_users                                8
dc_histogram_data                       9
dc_rollback_segments                   12
dc_histogram_data_values               44
dc_histogram_defs                      60
dc_segments                           117
dc_objects                            179
dc_object_ids                         195
```

可以看到，在 9*i* 版本的数据库中，字典缓存包含的都是 dc_xxx 的对象，我们知道 DC 是 Dictionary Cache 的缩写，也就是字典缓冲。行缓冲（Row Cache）也叫字典缓冲，这是绝大多数 DBA 在学习 Oracle 时就能学到的知识，而且很多培训老师也会告诉你字典缓冲为什么叫做 Row Cache，因为缓冲是以行为单位组织的，这里要区别于 DB Cache，DB Cache 是以块（Block）为单位的。有些知识更为渊博的老师还会告诉你，数据字典的缓冲管理和普通的表完全不同，字典缓冲表不使用 DB Cache，而使用字典缓存。

这个知识点似乎很正确，当年带我入门 Oracle 的老师是一个 40 多岁的香港人，已经是这个行业的资深人士了，对 Oracle 原理的研究也比较深。他当时就是这么教我的，我也理所当然地把这个知识点当成真理，又传授给很多人。在近 20 年前，几乎没有这方面的资料，因此我也没有去认真地求证。随着这些年对 DB Cache 和共享池研究的深入，我终于发现这个观点是错误的。字典缓存绝对不是纯粹的数据字典表的缓冲。数据字典表和普通的表没有不同，其数据块的缓冲也是相同的。而字典缓存是经过组织的，用于数据库运行中 SQL 解析、权限控制等用途的内部数据结构，是一种字典表的内存视图。如果 Oracle 在执行某个 SQL 的时候，为了 SQL 解析，需要访问一些字典表，从中获取一些数据，那么它会通过一些递归调用 SQL 来完成这些事情。这些 SQL 和普通 SQL 一样，需要将数据块从系统表空间中读取到 DB Cache 中，然后从 DB Cache 中获得 SQL 所需要的行数据。为了减少递归 SQL 的执行开销，这些行数据被缓存在共享池中，这个缓存就是字典缓存。

问题似乎明朗化了，字典缓存不是简单的数据字典表的缓冲，而是数据字典的缓冲。从更本质的角度讲，字典缓存其实是一种 Oracle 内部使用的数据结构，是基于数据字典表数据构建的，用于 SQL 解析等操作使用的数据结构。由于数据字典的数据十分庞大，Oracle 无法将所有需要的字典信息全部载入内存，因此对字典缓存的管理也采用了 LRU 的算法。字典缓存中保留了当期活跃的数据，一些不常用的数据将会被置换出去。

3.1.4　库缓存和游标

库缓存（library cache）也是共享池的一个子堆。和行缓存相比，库缓存及其存放的数据都更为复杂。在分析库缓存的内部结构之前，首先要了解一下库缓存中有哪些主要的对象。库缓存中的对象类型包括 Cursor、Table、Index、Cluster、View、Synonym、Sequence、Procedure、Function、

Package、Package Body、Trigger、Type、Type Body、Object、User、Database Link、Pipe、Table Partition、Index Partition、LOB、Library、Directory、Queue、Index-Organized Table、Java Source、Java Class、Java Resource、Java JAR、Table Subpartition、Index Subpartition、LOB Partition、LOB Subpartition、Summary、Dimension、Stored Outline 等。

任何一个对象类型肯定属于某个命名空间（namespace）。库缓存的对象可以在 V$DB_OBJECT_CACHE 中找到，这个视图基于 Oracle 内部结构的系统视图 X$KGLOB。下面的结果来自于 10.2.0.4 版本的数据库。

```
SQL>SELECT DISTINCT NAMESPACE FROM V$DB_OBJECT_CACHE;
NAMESPACE
----------------------------
CURSOR
INDEX
PUB_SUB
RSRC PLAN
SUBSCRIPTION
TABLE/PROCEDURE
APP CONTEXT
RSRC CONSUMER GROUP
JAVA RESOURCE
JAVA SHARED DATA
TRIGGER
JAVA SOURCE
RULESET
CLUSTER
INVALID NAMESPACE
BODY
```

讨论库缓存是一件十分复杂的事情，在库缓存中我们最为关心的对象是游标，因此首先来看一下游标的基本结构。一个游标的结构包括父游标和子游标，每个完整的游标必须包含一个父游标，并且至少包含一个子游标。我们知道每个 SQL 都对应相应的游标。但是在一个大型的系统中，库缓存可能有几千兆字节，Oracle 如何快速地找到某条 SQL 对应的游标呢？遍历整个库缓存、逐条比对 SQL 文本肯定不是一个好方法。Oracle 采用了十分简单而实用的方法来定位游标，对 SQL 的文本做散列计算，会得到一个散列值（HASH_VALUE），通过该散列值就可以找到相应的游标了。由于散列值是根据游标名称的每个字符进行散列计算得到的（对于 SQL 来说，游标的名称就是 SQL 文本本身），而 ASCII 字符大小写的内码是不同的，因此 select * from dual 和 SELECT * FROM DUAL 这两条语义完全相同的 SQL 将会生成不同的散列值，前者的散列值是 942515969，而后者的散列值是 3991932091，那么这两条 SQL 在 Oracle 数据库内部也就只能算是不同的语句了。这也是以前我们常说的编写 SQL 要注意大小写及空格的主要原因，相关示例如下。

```
SQL>  select hash_value,sql_text from v$sql where UPPER(SQL_TEXT) LIKE '%FROM DUAL';
HASH_VALUE                                                         SQL_TEXT
------------ ----------------------------------------------------------------
3991932091                                                SELECT * FROM DUAL
 942515969                                                select * from dual
3512579774                                                select * from dual
```

每个父游标包含一个 KGLHD、一个 KGLOB、一个或者多个 KGLNA。KGLOB 指向子堆（每个父游标至少包含一个 Heap 0，里面存放环境、状态和绑定变量的信息）。KGLHD 是每个游标的入口，其结构如代码清单 3-3 所示。

代码清单 3-3

```
struct kglhd {
  HASH Bucket 上的链接指针;
  关联对象列表指针;
  lock 持有者列表指针;
  lock 等待着列表指针;
  临时锁列表指针;
  Pin 拥有者列表指针;
  Pin 等待者列表指针;
  标识;
  Pin 实例锁状态;
  等待释放的 HANDLE 列表指针;
  Invalidation 实例锁状态;
  当前锁状态(KGLM0/N/S/X);
  当前 PIN 状态(KGLM0/S/X);
  库缓存的名字的指针（指向 KGLNA）;
  NAMESPACE ID;
  指向 kglob 的指针;
  Load/reload 计数;
  Invalidation 计数;
  执行次数计数;
  Pin 临时列表;
  子闩锁 ID;
  具有依赖关系的 handle 的链表指针;
  依赖列表指针;
}
```

上面的 KGLHD 结构是基于 Oracle 9*i* 的，由于 Oracle 在游标方面的优化力度很大，每个版本在性能上都有质的提升，因此各个版本中这个结构的变化也很大。不过其主要的结构仍然不变，我们还是可以从 9*i* 版本的结构中看出一些开发者的设计思想。KGLHD 结构包含了游标的一些最为基础的信息，通过 KGLHD 可以找到和这个游标相关的所有的关联对象，并且游标的一些统计信息也包含在这个结构中。在 KGLHD 中，有一个指向 KGLHD 的指针和一个指向 KGLNA 的指针，其中 KGLNA 存放的是这个库缓存的名字（对于 SQL，名字就是其本身，对于命名的 PL/SQL 对象，名字就是库缓存的名字）。KGLOB 则存放了库缓存的对象信息，其结构如代码清单 3-4 所示。

代码清单 3-4

```
struct kglob { /* 9201 struct kglob */
  指向 kglhd 的指针;
  当前 load lock 列表指针;
  等待 load lock 列表指针;
  依赖对象列表的指针;
  当前 load lock 的状态(KGLM0/X) ;
  FLAG;
  特殊状态，由库缓存闩锁保护;
```

```
    本状态的 BitMask 定义
        1 //  valid/无授权错误
        2 //  valid/有授权错误
        3 //  valid/有编译错误
        4 //  valid/未授权
        5 //  invalid/未授权
        6 //  invalid/未授权但保留时间戳
    KGLOB 的类型
      BitMask
        0 //  cursor
        1 //  index
     ...
     ...
    指针指向 KGLOB SUBHEAP[16];
    和 HEAP0 有关的信息区域的指针;
}
```

每个 KGLOB 包含的子堆可能的情况如下：

```
Heap # Description
0      Object
1      Source
2      DIANA
3      PCODE
4      MCODE
5      Errors
6      SQL Context
7      Free
8      Subordinate Heap
9      Subordinate Heap
10     Subordinate Heap
11     Subordinate Heap
```

对于库缓存结构的描述，Julian Dyke 大师 PPT 中的一张关于父游标的结构图描述得很清楚，这里老白偷个懒，引用一下大师的图示，如图 3-1 所示。

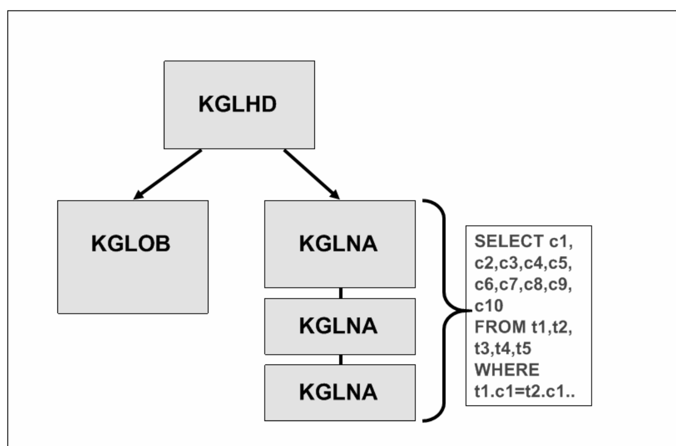


图　3-1

看了上面老白描述的 KGLHD 和 KGLOB 结构，再来看这张图就十分清晰了。对于一个游标来说，父游标包含句柄（KGLHD）、对象结构（KGLOB）和名字结构（KGLNA）。每个父游标在 V$SQLAREA 中有一条记录。在 X$KGLOB 中， KGLHDPAR = KGLHDADR 的记录就是父游标的。每个父游标至少有一个子游标，子游标中包含：环境信息、统计信息、绑定变量、执行计划等。每个子游标包含一个 KGLHD、一个 KGLOB 和 SUBHEAP。在每个子游标中，包含一个 Heap 6，里面存放的是执行计划。

下面继续引用 Julian Dyke 的图例，如图 3-2 所示。

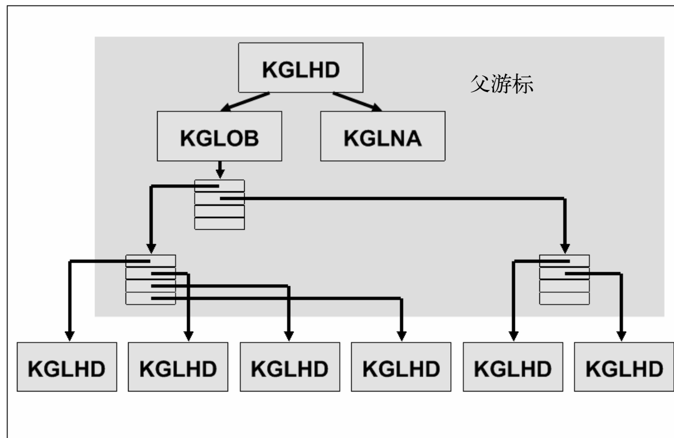


图 3-2

这张图明确地显示了父游标和子游标之间的关系。父游标的 KGLOB 中的 KGLDA 里的某个列表中存放了子游标的句柄（KGLHD）指针。对于子游标的结构，Dyke 也有一张很棒的图示，如图 3-3 所示。

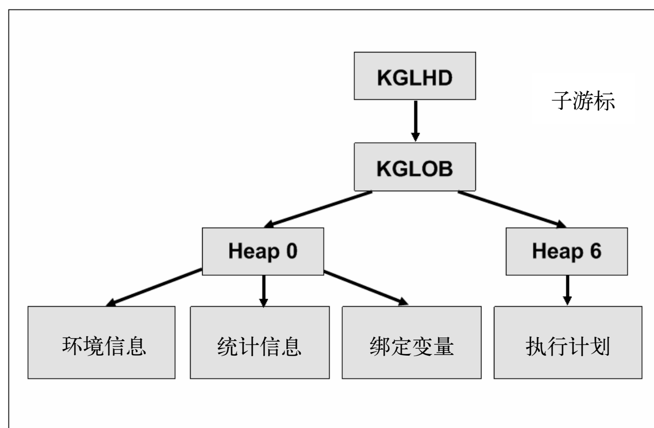


图 3-3

我们看到 Heap 0 中包含一些环境信息、统计信息和绑定变量的信息，Heap 6 中包含了这个 SQL 的执行计划。

本节的最后，我们通过一个实际的库缓存转储来验证前面讨论的内容，首先看一下父游标的转储信息。


```
LIBRARY OBJECT HANDLE: handle=7b9bc980
      name=SELECT abc,pos from Element  WHERE pid=:1 and namespace=:2 order by pos
      hash=931b7ee9 timestamp=12-21-2011 19:19:06
      namespace=CRSR flags=RON/KGHP/TIM/PN0/MED/[50010000]
      kkkk-dddd-llll=0000-0001-0001 lock=N pin=0 latch#=1
      lwt=0x7b9bc9b0[0x7b9bc9b0,0x7b9bc9b0] ltm=0x7b9bc9c0[0x7b9bc9c0,0x7b9bc9c0]
      pwt=0x7b9bc9e0[0x7b9bc9e0,0x7b9bc9e0] ptm=0x7b9bca70[0x7b9bca70,0x7b9bca70]
      ref=0x7b9bc990[0x7b9bc990, 0x7b9bc990] lnd=0x7b9bca88[0x7b9bca88,0x7b9bca88]
        LIBRARY OBJECT: object=7b9bc5a8
        type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
        CHILDREN: size=16
        child#    table reference   handle
        ------ -------- --------- --------
             0 7b9bc808  7b9c4e40 7b9c4c08
```

LIBRARY OBJECT 的 HANDLE（kglhd）的地址是 7b9bc980，名字就是 SELECT 语句本身。散列值是 0x931b7ee9。LIBRARY OBJECT（kglob）的地址是 7b9bc5a8，其中的子游标列表的大小是 16，这是因为 KGLOB 数据结构中定义了固定大小的列表，因此如果列表中数据项不足 16 会占用一个列表，超过 16 个会级联多个列表。这个游标只有一个子游标，其 KGLHD 的地址是 7b9c4c08，通过该地址我们找到了这个子游标：

```
LIBRARY OBJECT HANDLE: handle=7b9c4c08
      namespace=CRSR flags=RON/KGHP/PN0/[10010000]
      kkkk-dddd-llll=0000-0041-0041 lock=N pin=0 latch#=1
      lwt=0x7b9c4c38[0x7b9c4c38,0x7b9c4c38] ltm=0x7b9c4c48[0x7b9c4c48,0x7b9c4c48]
      pwt=0x7b9c4c68[0x7b9c4c68,0x7b9c4c68] ptm=0x7b9c4cf8[0x7b9c4cf8,0x7b9c4cf8]
      ref=0x7b9c4c18[0x7b9c4e40, 0x7b9c4e40] lnd=0x7b9c4d10[0x7b9c4d10,0x7b9c4d10]
        LIBRARY OBJECT: object=7b9c4830
        type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
        DEPENDENCIES: count=1 size=16
        ACCESSES: count=1 size=16
        TRANSLATIONS: count=1 size=16
        DATA BLOCKS:
        data#     heap  pointer status pins change
        ----- -------- -------- ------ ---- ------
            0 7b9c4b48 7b9c4530 I/P/A     0 NONE
            6 7b9c4950 7b9bcea8 I/-/A     0 NONE
        ----------------------------------------
```

可以看到，这个子游标的 KGLOB 指向了两个子堆，SUBHEAP 0 和 SUBHEAP 6，一个存放环境信息，另一个存放执行计划。

3.2　共享池和游标

游标是共享池中最为重要的组件之一，如果仅仅讨论共享池而不涉及游标，那么我们就很难

理解共享池的大部分工作原理。另外，优化共享池在多数情况下就是优化游标并发访问的性能。本节主要讨论共享池和游标的一些基本原理。

3.2.1　游标与游标共享

游标共享是共享池的重要功能之一，它可以提高共享池的使用效率，减少 SQL 解析的开销，从总体上提高 SQL 执行的效率。如果一条 SQL 能够被解析一次，执行多次，那么就可以达到比较好的效果，减少分析的开销，这是 Oracle SQL 共享的最高目标。

要实现游标共享，首先要具备一定的机制，也就是说游标的一些执行结构不能存放在程序的私有空间里，而需要存放在共享内存中。共享池中的库缓存就是实现这种共享机制的载体。一个可共享的游标，其可共享的部件是存放在库缓存中的，这样就实现了不同 SESSION 共享同一 SQL。

满足了上述条件后，下一步就要来判断哪些 SQL 是可以共享的。最简单的判断方法就是：SQL 语句完全相同的 SQL 是可以共享的。如何来判断 SQL 完全相同呢？如果能对该 SQL 的语义语法进行全面解析，通过最终分解出的 TOKEN 来进行比较是最好的，这样能够对 SQL 进行全面的识别。但是这种识别方式的开销很大，Oracle 采取了一种十分巧妙的方法来分辨不同的 SQL。通过对 SQL 的文本进行计算，生成一个散列值，如果散列值不同，那么 SQL 肯定不同；如果散列值相同，就可能是可以共享的 SQL。这种机制的实现十分简单，比较相同 SQL 的开销也非常小，不过它对 SQL 的书写要求较高，对于大小写、空格等有严格的要求，如果不符合要求，即使语法语义完全相同的 SQL，Oracle 也会认为是不同的。

基于上述原理，Oracle 判断游标共享的第一个原则是，可共享的游标的 SQL 文本必须完全相同。一个游标在执行前，首先对其文本计算散列值，然后通过该散列值在 HASH Bucket 上查找，如果找到了相同的游标，而且该游标的所有对象（包括 SUBHEAP）都是可用的（VALID），那么这个 SQL 在执行的时候，就可以使用共享的游标。如果某些对象已经被换出（AGEOUT），那么这个游标就需要进行软分析，将丢失的部分补充完整才能执行。

如果两条 SQL 的文本完全相同，是不是就一定能够共享呢？答案是否定的。比如，SCOTT 和 TIGER 这两个 SCHEMA，其下都有名为“TT”的表。如果在这两个用户下都执行 select 1 from tt where rownum<2，则两条 SQL 所访问的表是不同的，因此它们是不应该共享的。Oracle 在这种情况下是怎么处理的呢？首先，由于 SQL 文本完全相同，所以这两条 SQL 具有相同的 SQL_ID 和散列值，被认为是相同的 SQL，在 V$SQLAREA 中可以看到如下结果：

```
SQL_ID         ADDRESS   SQL_TEXT                           VERSION_COUNT
cpjnybv7021rv   1F7059E0   select 1 from tt where rownum<2     2
```

这条 SQL 的 version count 是 2，即存在两个子游标。接下来我们在 V$SQL 中看到如下结果：

```
SQL_ID         ADDRESS       SQL_TEXT
------------------------------------
cpjnybv7021rv   1F7059E0        select 1 from tt where rownum<2
cpjnybv7021rv   1F7059E0        select 1 from tt where rownum<2
```

可以看出，这两条 SQL 被认为是相同的，但是 cpjnybv7021rv 包含两个子游标。为什么会产生两个子游标呢？通过 v$sql_shared_cursor 可以看到：

```
SQL> select sql_id,address,child_address,child_number,translation_mismatch from
  2  v$sql_shared_cursor where sql_id='cpjnybv7021rv';

SQL_ID        ADDRESS  CHILD_AD CHILD_NUMBER T
------------- -------- -------- ------------ -
cpjnybv7021rv 1F7059E0 231974B8            0 N
cpjnybv7021rv 1F7059E0 232BC270            1 Y
```


不难发现，第一个子游标在这个视图中的所有 mismatch 都是 N，而第二条 SQL 由于 translation_mismatch 导致不能共享，其原因是在执行 translation 时发现相关的对象不同。通过库缓存转储得到如下结果（alter system set events 'immediate trace name library_cache level 10';）：

```
Bucket 67323:
  LIBRARY OBJECT HANDLE: handle=1f7059e0 mutex=1F705A94(2)
  name=select 1 from tt where rownum<2
  hash=1cd693ce0a964189cac69e5ece0106fb timestamp=12-11-2007 16:18:05
  namespace=CRSR flags=RON/KGHP/TIM/KEP/PN0/SML/KST/DBN/MTX/[120100d4]
  kkkk-dddd-llll=0001-0001-0001 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
  lwt=1F705A3C[1F705A3C,1F705A3C] ltm=1F705A44[1F705A44,1F705A44]
  pwt=1F705A20[1F705A20,1F705A20] ptm=1F705A28[1F705A28,1F705A28]
  ref=1F705A5C[1F705A5C,1F705A5C] lnd=1F705A68[232C803C,1F705164]
    DEPENDENCY REFERENCES:
    reference latch flags
    --------- ----- -------------------
     206b0c68     0 [60]
     20642c5c     0 [60]
    LIBRARY OBJECT: object=206b138c
    type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
    CHILDREN: size=16
    child#    table reference   handle
    ------ -------- --------- --------
         0 206b1318  206b0fcc 231974b8
         1 206b1318  206b112c 232bc270
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 2316d4bc 206b1424 I/P/A/-/-    0 NONE   00
  Bucket 67323 total object count=1
```

从 TRACE 上我们可以看出，这个父游标的文本就是 select 1 from tt where rownum<2，它包含了 2 个子游标，其中一个子游标的句柄（KGLHD）的地址是 231974B8，其详细信息如下：

```
LIBRARY OBJECT HANDLE: handle=231974b8 mutex=2319756C(0)
  namespace=CRSR flags=RON/KGHP/PN0/[10010000]
  kkkk-dddd-llll=0000-0041-0041 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
  lwt=23197514[23197514,23197514] ltm=2319751C[2319751C,2319751C]
  pwt=231974F8[231974F8,231974F8] ptm=23197500[23197500,23197500]
  ref=23197534[206B0FCC,206B0FCC] lnd=23197540[23197540,23197540]
    CHILD REFERENCES:
    reference latch flags
    --------- ----- -------------------
     206b0fcc     0 CHL[02]
```

```
LIBRARY OBJECT: object=206b0b2c
type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
DEPENDENCIES: count=1 size=16
dependency#    table reference   handle position flags
----------- -------- --------- -------- -------- -------------------
          0 20759070  20758d70 230c63b8       14 DEP[01]
READ ONLY DEPENDENCIES: count=1 size=16
dependency#    table reference   handle flags
----------- -------- --------- -------- -------------------
          0 206b0ee8  206b0c68 1f7059e0 /ROD/KPP[60]
ACCESSES: count=1 size=16
dependency# types
----------- -----
          0 0009
TRANSLATIONS: count=1 size=16
original    final
-------- --------
230c63b8 230c63b8
DATA BLOCKS:
data#     heap  pointer    status pins change whr
----- -------- -------- --------- ---- ------ ---
    0 23202038 206b0c7c I/-/A/-/-    0 NONE   00
    6 20758ca4 201a185c I/-/A/-/-    0 NONE   00
```

这个子游标的依赖关系列表中有一个对象的句柄是 230c63b8，我们再来看看 230c63b8 对应的对象到底是什么。

```
Bucket 99929:
  LIBRARY OBJECT HANDLE: handle=230c63b8 mutex=230C646C(0)
  name=SCOTT.TT
  hash=dcfddf221c9799c3b07c3d16af4b8659 timestamp=12-11-2007 16:16:45
  namespace=TABL flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0701-0701 lock=N pin=0 latch#=1 hpc=0002 hlc=0002
  lwt=230C6414[230C6414,230C6414] ltm=230C641C[230C641C,230C641C]
  pwt=230C63F8[230C63F8,230C63F8] ptm=230C6400[230C6400,230C6400]
  ref=230C6434[230C6434,230C6434] lnd=230C6440[232B9660,1F6575D8]
    DEPENDENCY REFERENCES:
    reference latch flags
    --------- ----- -------------------
     20478ce0     2 DEP[01]
     20758d70     0 DEP[01]
    LOCK OWNERS:
        lock     user  session count mode flags
    -------- -------- -------- ----- ---- -----------------------
    20f9e5e4 2373238c 23733654     0 N    [4000]
    20fc86cc 2373238c 2373238c     0 N    [4000]
    LIBRARY OBJECT: object=20758884
    type=TABL flags=EXS/LOC[0005] pflags=[0000] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 2324da64 2075891c I/-/A/-/-    0 NONE   00
        8 20758aac 206b7298 I/-/A/-/-    0 NONE   00
        9 20758b44 204cf62c I/-/A/-/-    0 NONE   00
       10 20758b94 2073e4e0 I/-/A/-/-    0 NONE   00
  Bucket 99929 total object count=1
```

从名称上可以看出这个对象是表 SCOTT.TT，再来看看第二个子游标 232bc270 的信息。

```
LIBRARY OBJECT HANDLE: handle=232bc270 mutex=232BC324(0)
  namespace=CRSR flags=RON/KGHP/PN0/[10010000]
  kkkk-dddd-llll=0000-0041-0041 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
  lwt=232BC2CC[232BC2CC,232BC2CC] ltm=232BC2D4[232BC2D4,232BC2D4]
  pwt=232BC2B0[232BC2B0,232BC2B0] ptm=232BC2B8[232BC2B8,232BC2B8]
  ref=232BC2EC[206B112C,206B112C] lnd=232BC2F8[232BC2F8,232BC2F8]
    CHILD REFERENCES:
    reference latch flags
    --------- ----- -------------------
     206b112c     0 CHL[02]
    LIBRARY OBJECT: object=20642b20
    type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
    DEPENDENCIES: count=1 size=16
    dependency#    table reference   handle position flags
    ----------- -------- --------- -------- -------- -------------------
              0 2059d22c  2059cf2c 1f708c18       14 DEP[01]
    READ ONLY DEPENDENCIES: count=1 size=16
    dependency#    table reference   handle flags
    ----------- -------- --------- -------- -------------------
```

从中找出依赖的对象 1f708c18，其对应的对象为：

```
Bucket 68671:
  LIBRARY OBJECT HANDLE: handle=1f708c18 mutex=1F708CCC(0)
  name=T2.TT
  hash=b6efdd23a914ea2ea1b8ffeb7d170c3f timestamp=12-11-2007 16:16:15
  namespace=TABL flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0701-0701 lock=N pin=0 latch#=2 hpc=0002 hlc=0002
  lwt=1F708C74[1F708C74,1F708C74] ltm=1F708C7C[1F708C7C,1F708C7C]
  pwt=1F708C58[1F708C58,1F708C58] ptm=1F708C60[1F708C60,1F708C60]
  ref=1F708C94[1F708C94,1F708C94] lnd=1F708CA0[1F705910,1F708B48]
    DEPENDENCY REFERENCES:
    reference latch flags
    --------- ----- -------------------
     207a6154     2 DEP[01]
     2059cf2c     0 DEP[01]
    LOCK OWNERS:
        lock     user  session count mode flags
    -------- -------- -------- ----- ---- ------------------------
    2125a5dc 2373491c 23733654     0 N    [4000]
    212345d0 2373491c 2373491c     0 N    [4000]
    LIBRARY OBJECT: object=2059ca40
    type=TABL flags=EXS/LOC[0005] pflags=[0000] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 2324abc0 2059cad8 I/-/A/-/-    0 NONE   00
        8 2059cc68 2059c8fc I/-/A/-/-    0 NONE   00
        9 2059cd00 2059c10c I/-/A/-/-    0 NONE   00
       10 2059cd50 2059bd14 I/-/A/-/-    0 NONE   00
  Bucket 68671 total object count=1
```

这个对象是 T2.TT，这两张表是完全不同的，所以 SQL 不能共享。除了这种情况，还有哪些不可共享的原因呢？从 v$sql_shared_cursor 的字段中就可以看出不可共享的各种原因。

另外一种典型的 SQL 不能共享的情况是两个 SQL 的执行计划不同。如果两个 SQL 相同，但是由于参数不同，必须使用不同的执行计划，那么最好不要共享这种 SQL。因为 SQL 共享带来的好处可能只是执行了错误的执行计划的几百分之一。从这方面来看，我们也不能片面地强调 SQL 共享，而忽略了由于 SQL 共享带来的问题。

举个简单的例子。TA 表有个字段是 STATUS，其中 99%的值都是 END，只有 1%的字段是 BEGIN。我们的大多数程序都是每次读取值为 BEGIN 的行，然后处理，处理结束后 STATUS 变成 END。只有少量的统计操作需要统计 END 值的行数量。这时如果我们可以使用柱状图，优化器就能够做出判断并使用合理的执行计划。这种情况下，不能使用绑定变量。如果使用了绑定变量，那么对于早期版本，优化器就不会使用柱状图，而会使用默认的选择性值来判断；如果是 9*i* 或者更高版本的数据库，绑定变量窥探技术可以通过使用柱状图来选择较好的执行计划，但是它只在 SQL 第一次被执行时，执行硬解析的时候进行，由于后面的所有 SQL 都使用了绑定变量，会被认为是安全的，这条 SQL 会使用共享的游标，因此就不会进行窥探。这条 SQL 可能有两种执行计划，索引扫描或者全表扫描，至于选择哪种执行计划，则完全取决于执行硬解析的那条 SQL 的绑定变量的值，而无法由优化器做出最佳的选择。在这种情况下，共享 SQL 的代价就太大了。所以 SQL 共享是优化的手段，而不是优化的目标，千万不能为了优化而优化。对于不同的变量值，希望使用不同的执行计划，如果选错了执行计划，会大幅度增加 SQL 的开销，那么就要慎用绑定变量了。这个时候可能不使用绑定变量对系统整体性能的改善最有利。在 9*i* 版本中，默认情况下表分析是不采集柱状图的，而在 10*g* 版本中，柱状图的采集是默认的。

不过 10*g* 和 9*i* 的绑定窥探技术还是存在一定的局限性。对于使用了绑定变量的 SQL，绑定窥探往往无法达到较好效果。因此 Oracle 11*g* 引入了一种新的游标共享机制 ACS（Adaptive Cursor Sharing，自适应游标共享），首先引入 ACS 的稳定版本是 11.1.0.6。ACS 在共享 SQL 和 SQL 执行时的资源消耗之间实了更好的平衡，其基本思想就是对于使用了绑定变量的 SQL，不会盲目地使用一个共享的执行计划，而是根据其绑定变量值的选择性的不同，分为几个组，每组使用不同的执行计划。ACS 的出现，使绑定变量的使用没有了顾虑，可以有效地提高此类 SQL 的效率。

但是，ACS 也存在一些副作用。首先由于每次 SQL 执行都需要分析绑定变量，这增加了 SQL 解析的开销。同时，它也加大了游标不能共享的机会，增加了硬解析的比例。另外一个游标中不能共享的版本的数量也会大大增加，这样就影响了库缓存的性能。

另外，由于目前 ACS 还算是一种新技术，肯定存在一些不完善的地方，在高并发的系统中，也可能会出现一些 Bug。因此我们在使用 ACS 这个新功能的时候，还是要十分注意，ACS 可能会导致一个父游标下面有大量的子游标无法共享，实际上这些游标都是可以共享的。如果碰到类似的情况，那么就必须关闭 ACS，使用 10*g* 或者以前版本的游标共享机制了。

3.2.2　游标与 SQL 的执行

搞清楚 SQL 执行的原理一直是绝大多数 DBA 所希望的，这些年来老白也一直在做这方面的

研究。不过由于 SQL 执行涉及太多 Oracle 底层的原理，而且关联的知识点十分广泛，因此老白也仅仅是管中窥豹，略知一二。根据 Oracle 官方的说法，SQL 语句的执行有以下步骤：

- ❑ Syntactic，语法检查。
- ❑ Semantic，确认所有对象都存在并且可以访问。
- ❑ View Merging，进行查询重写优化。
- ❑ Statement Transformation，将复杂的查询分解。
- ❑ Optimization，确定访问方式，选择优化策略。
- ❑ QEP Generation，形成执行计划。
- ❑ QEP Execution，运行执行计划。

上面的 QEP 是查询执行计划的英文简称。在这七个步骤中，前六步就是我们通常所说的解析（PARSING），第七步就是通常所说的执行（EXECUTION）。正如前面几节所述，为了不重复解析相同的 SQL 语句，在第一次解析之后，Oracle 将 SQL 语句存放在库缓存中。这块位于系统全局区域 SGA 的共享池中的内存可以被所有的数据库会话共享。因此，一条 SQL 语句在执行时，如果和之前执行过的语句完全相同，Oracle 就能很快获得已经被解析的语句以及最好的执行路径。Oracle 的这个功能大大提高了 SQL 的执行性能并节省了内存的使用。表 3-1 更加详细地描述了 SQL 的执行过程（表 3-1 来自于 Metalink 上的文档，老白将其翻译为中文了）。

表 3-1

<table>
<tr><td rowspan="5">PARSE</td><td colspan="3">SQL匹配，语法语义检查</td><td>通过对库缓存中对象的比对，进行匹配</td></tr>
<tr><td colspan="3">Query Transformation</td><td>对子查询、视图等进行重新组合和SQL改写</td></tr>
<tr><td rowspan="3">RBO——根据规则制定执行计划</td><td>CBO</td><td>判断对象访问的开销以及结果集的大小</td><td>每个对象都独立计算成本以及返回的结果集的大小</td></tr>
<tr><td>CBO——判断不同的连接顺序的不同开销</td><td>连接方式和连接顺序被通盘考虑，并且找到开销最小的连接方式</td><td>这个步骤里面包含了SQL执行计划的优化</td></tr>
<tr><td>产生执行树</td><td>执行树被生成后放在库缓存里，当SQL执行的时候，被用来驱动查询</td><td></td></tr>
<tr><td colspan="4">EXECUTE</td><td>分配绑定变量需要的内存空间，绑定变量的值实现绑定。使用上一步产生的执行计划执行SQL</td></tr>
<tr><td colspan="4">FETCH</td><td>SELECT操作比普通的SQL多了一个FETCH步骤，在这个步骤中，实际的DB BLOCK的访问才会产生。在这个阶段，将剔除不需要的数据，把结果放入结果集，传输给客户端</td></tr>
</table>

在 10046 Trace 里可以很清楚地看到这些 SQL 执行的步骤。不过实际上 SQL 执行并不像上述那样简单明了。这些步骤之间可能会产生交叉和重叠，且部分操作可能提前或滞后。比如，对于

使用了绑定变量情况下的绑定窥探，要实现窥探，分析中的优化步骤可能延后到执行阶段进行，因为只有在执行阶段，绑定变量的值才是明确的。

要执行一条 SQL，并且尽可能使用共享的游标，Oracle 采用如图 3-4 所示的判断流程（来自 Oracle 官方的说明文档）来共享游标。

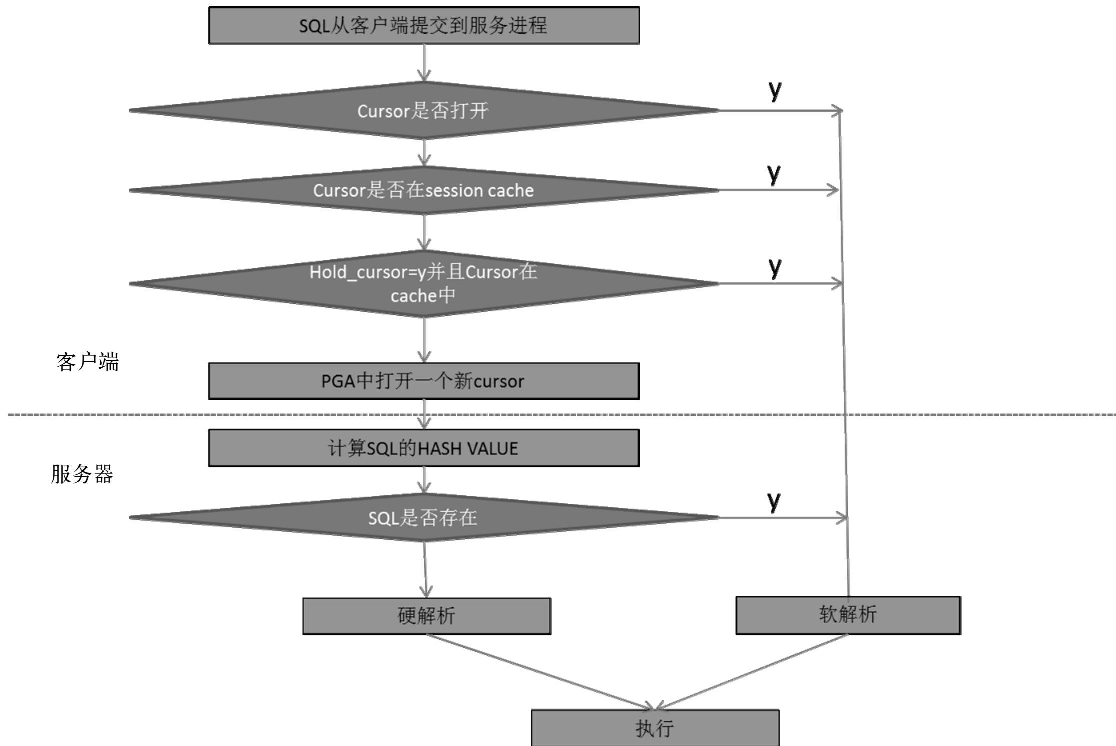


图　3-4

不同类型 SQL 的执行步骤也有所不同。比如，INSERT 语句中就没有 FETCH 这个步骤，FETCH 是 SELECT 语句的独有步骤。记得前些年有人问老白一个问题，SELECT 语句执行的过程中，读数据块、从中查找数据是在 EXECUTE 阶段完成的，还是在 FETCH 阶段完成的呢？以老白做应用架构设计的经验来说，有些 SQL 在读取一个游标的时候，有可能不会把打开的游标从头到尾读取一遍，就因为某种原因结束了读取，甚至关闭游标。如果 SQL 执行过程中，在读取之前就已经遍历了所有的数据块，找出所有的结果数据，那么就会造成浪费。因此，在 FETCH 阶段才去访问数据才是最好的选择。

不过 Oracle 公司的官方资料并没有这方面的描述，因此老白想到了一个验证这个观点的方法。首先我们创建一张表：

```
SQL> create table scott.testb as select * from dba_objects;
Table created.
```

为了得到准确的数据，需要重启数据库。重启之后，数据库的 DB Cache 是干净的，然后我们执行：

```
Set pause on
Select object_id from scott.testb;
```

使用 set pause on 是为了让 SQL*Plus 的 FETCH 操作不能立即完成。每次停顿等待屏幕输入时，FETCH 操作是停止的。然后在另外一个窗口，查询 V$BH 视图，查看 TCH 指标的变化：

```
SQL> COL OBJECT_NAME FORMAT A40 TRUNCATE;
SQL> COL SUBCACHE FORMAT A10 TRUNCATE;
SQL> Select decode(pd.bp_id,1,'KEEP',2,'RECYCLE',3,'DEFAULT',4,
  2  '2K SUBCACHE', 5,'4K SUBCACHE',6,'8K SUBCACHE',7,
'16K SUBCACHE',8,'32K SUBCACHE','UNKNOWN') subcache, bh.object_name
object_name,bh.blocks,tch from x$kcbwds ds,
x$kcbwbpd pd, (select /*+ use_hash(x) */ set_ds,
o.name object_name, count(*) BLOCKS,sum(tch)  tch
  3    4    5    6  from obj$ o, x$bh x where o.dataobj# = x.obj
  7  and x.state !=0 and o.owner# !=0
group by set_ds,o.name) bh where ds.set_id >= pd.bp_lo_sid
and ds.set_id <= pd.bp_hi_sid and pd.bp_size != 0 and ds.addr=bh.set_ds
order by subcache,object_name;

SUBCACHE   OBJECT_NAME                                  BLOCKS        TCH
---------- ---------------------------------------- ---------- ----------
DEFAULT    AQ$_QUEUES                                        2          2
DEFAULT    AQ$_QUEUES_CHECK                                  1          1
DEFAULT    AQ$_QUEUE_TABLES                                  1          1
DEFAULT    AQ$_QUEUE_TABLES_PRIMARY                          1          1
DEFAULT    DEF$_AQCALL                                       1          1
DEFAULT    DEF$_AQERROR                                      1          1
DEFAULT    REPCAT$_REPPROP                                   1          1
DEFAULT    SYS_IOT_TOP_10254                                 1          1
DEFAULT    SYS_IOT_TOP_50155                                 1          1
DEFAULT    TESTB                                           329         57
DEFAULT    XDB$CONFIG                                        6          6
DEFAULT    XDB$SCHEMA_URL                                    1          1

12 rows selected.

SUBCACHE   OBJECT_NAME                                  BLOCKS        TCH
---------- ---------------------------------------- ---------- ----------
DEFAULT    AQ$_QUEUES                                        2          2
DEFAULT    AQ$_QUEUES_CHECK                                  1          1
DEFAULT    AQ$_QUEUE_TABLES                                  1          1
DEFAULT    AQ$_QUEUE_TABLES_PRIMARY                          1          1
DEFAULT    DEF$_AQCALL                                       1          1
DEFAULT    DEF$_AQERROR                                      1          1
DEFAULT    REPCAT$_REPPROP                                   1          1
DEFAULT    SYS_IOT_TOP_10254                                 1          1
DEFAULT    SYS_IOT_TOP_50155                                 1          1
DEFAULT    TESTB                                           384        116
DEFAULT    XDB$CONFIG                                        6          6
```

```
DEFAULT    XDB$SCHEMA_URL                                          1          1

12 rows selected.

SQL> /

SUBCACHE   OBJECT_NAME                                        BLOCKS        TCH
---------- ---------------------------------------- ---------- ----------
DEFAULT    AQ$_QUEUES                                              2          2
DEFAULT    AQ$_QUEUES_CHECK                                        1          1
DEFAULT    AQ$_QUEUE_TABLES                                        1          1
DEFAULT    AQ$_QUEUE_TABLES_PRIMARY                                1          1
DEFAULT    DEF$_AQCALL                                             1          1
DEFAULT    DEF$_AQERROR                                            1          1
DEFAULT    REPCAT$_REPPROP                                         1          1
DEFAULT    SYS_IOT_TOP_10254                                       1          1
DEFAULT    SYS_IOT_TOP_50155                                       1          1
DEFAULT    TESTB                                                 399        136
DEFAULT    XDB$CONFIG                                              6          6
DEFAULT    XDB$SCHEMA_URL                                          1          1

12 rows selected.
```

可以看出，随着读取的进行，载入内存的 BLOCKS 和 TCH 都在增长。而未执行读取数据操作时，这些数据都是静止不动的。这个例子只能证明数据是随着读取的进行而被访问的，那么 Oracle 是否存在一些预读取之类的行为呢？答案是肯定的，为了提高读取数据的效率，Oracle 提供了预读取功能。通过下面几个参数可以控制预读取的行为。

```
_db_block_prefetch_limit
_db_block_prefetch_quota
_table_lookup_prefetch_size
```

其中，_db_block_prefetch_limit 和 _db_block_prefetch_quota 控制数据块预读的数量；_table_lookup_prefetch_size 控制数据行预读的数量，这个参数的默认值在 8.0 版本中为 10，9*i* 及后续版本中被加大为 40。对于大量读取数据的操作，我们一般以 BULK COLLECT 的方式来批量处理，从而提高读取的性能。此外，BULK COLLECT 配以合适的_table_lookup_prefetch_size 参数值也可以提高读取的性能。不过由于_table_lookup_prefetch_size 参数不能做会话级调整，因此这个参数的调整要十分慎重，一旦调整得不合理将会影响系统的性能。如果调整幅度太大，可能会由于过量的预读取而增加不必要的开销。

在 SQL 的执行过程中，如果这个游标是开放的，那么 SQL 的执行分析开销最小，不需要做任何解析，就可以直接执行了，这种情况下没有解析产生。如果执行的游标在 SESSION CACHE 中，虽然还是统计了一次软解析，但是和一般的软解析不同，它的开销很小，Tom 称之为软软解析（soft soft parse），就是用以区别于普通的软解析。

其实解析的过程是十分复杂的，绝对不是硬解析（hard parse）、软解析（soft parse）和软软解析（soft soft parse）这三种方式就可以概括的。比如，执行一条 SQL 的时候，如果该 SQL 在共享池中不存在，那么很简单，这就是硬解析，需要首先分配共享池空间，创建父游标的结构，

然后创建一个子游标。如果下一次再执行一条类似的 SQL，该 SQL 的父游标存在，经过检查发现子游标是可以共享的，而且这个子游标的所有关联对象在共享池中都存在，那么就可以马上执行了。这就是我们所说的软软解析，其实这也是软解析的一种。还有一种情况，如果子游标是不可共享的，那么我们就需要创建一个新的子游标（对于 SQL 来说，又增加了一个版本，Version），解析执行计划，然后执行。这也被记录为一次软解析，它的开销明显比刚才的软软解析高出很多。其实我们还经常碰到另外一种情况，当我们找到某个子游标的时候，发现该游标相关的关联对象的数据不完整，那么就必须重新生成这个子游标才能够执行 SQL，这也是一种软解析。为什么某个子游标会不完整呢？共享池是一种采用 LRU 机制的共享缓冲，当共享池空闲空间不足的时候，就会换出某些对象，从而腾出新的空间给需要的会话。共享池在释放内存的时候，首先释放那些没有上锁的对象，如果所有没上锁的对象都被释放了，空间还是不足，才会释放带有空锁的对象，而被锁住的对象是不能释放的。

从 SQL 执行的原理来看，尽可能减少硬解析，甚至减少软解析都可能给系统带来性能的提升。要想减少硬解析和软解析，就要让 SQL 能够尽可能地在共享池中多保存一段时间，并且那些执行十分频繁的 SQL 要尽可能保存在共享池中。保持共享池有足够的空间存放这些对象是十分重要的，因此在共享内存自动管理的情况下，当 Buffer Cache 能够保持足够高的命中率时，Oracle 总是尽可能地扩展共享池，这样也导致了共享池十分庞大。

适当保持足够大的共享池对于 OLTP 系统来说是十分重要的。只有共享池足够大了，才能够让游标的所有相关数据尽可能多地保存在共享池中不被换出。实际上，一套应用系统需要配置多大的共享池，和应用的总体架构以及应用系统的特点是息息相关的。对于一些负载十分高、每秒事务处理数量达到 1000 左右的系统来说，可能 5GB 左右的共享池就足够了，而对于另外一套并发处理量小 10 倍、每秒事务数不超过 20 个的系统来说，可能 8GB 共享池还经常出现严重的争用。因此我们在分配共享池容量的时候，一定要从实际应用系统出发，绝对不能带着有色眼镜看问题。

在互联网上还存在着一个观点，就是共享池不宜过大，过大的共享池反而会影响性能。这个观点看似有点道理，因为如果共享池太大，搜索对象所需要的时间也会增加，这就存在降低性能的可能性。不过反过来想想，这种观点似乎也站不住脚。对于某个确定的 Oracle 版本来说，各种 Hash 链的 Bucket 数量是固定的，搜索某个对象的速度取决于串在 Hash 链上的对象的数量，如果加大了共享池，导致大量的共享池空间是没有使用的，那么这些空闲的内存并不会增加搜索共享池对象的开销。如果由于增加了共享池的内存，导致更多的共享池对象被放入共享池中，从而使 Hash 链变长，这确实可能增加检索 Hash 链的开销。但是和不停地释放共享池、分配共享池所产生的开销相比，这些开销可能小很多了。因此共享池分配多少合适，并不是一个简单的问题，需要和应用特点相结合去分析，有时候确实需要经过一段时间的验证才能够得到合理的答案。幸运的是，从 9*i* 版本开始，我们可以动态分配共享池空间，不需要因修改共享池参数而重启数据库实例了。

这并不是说可以无限制地配置共享池。共享池配置还是以适当为好，过大的共享池实际上对提高并发访问性能没什么益处，反而会由于堆管理成本增加而导致共享池性能略有下降。加大共

享池不是万应良药，在大多数情况下，共享池的冲突和游标相关的参数设置与应用程序未合理使用绑定变量有关，从源头上解决问题才是最佳的。

3.2.3　游标共享和绑定变量

从前面的讨论我们已经了解了 SQL 共享的重要性。SQL 共享可以提高共享池的效率，减少 SQL 的解析成本。而实现 SQL 共享最好的技术就是使用绑定变量，使 SQL 代码一致，从而减少解析带来的开销。

不过绑定变量是让人爱恨交加的技术。在 CBO 优化器功能不是很强的年代，很多系统采用 rule 提示来强制使用 RBO 优化器，在这种情况下绑定变量的缺点还没有那么明显，大不了不用 CBO 优化器罢了。但在 Oracle 8*i* 或者以前的版本中，对于绑定变量的 CBO 执行计划的产生采用固定的方式，因此对于那些不均匀列上面的条件，产生的估算结果往往准确性欠佳。

正是因为这个原因，Oracle 9*i* 才引入了绑定变量窥探技术（Bind Value Peeking，我印象中是 9.2，不知道是否确切）。在没有这项技术之前，使用绑定变量后，执行计划的产生要靠默认的选择性判断，这种判断往往会出现严重的偏差。加了这项技术后，当 SQL 进行硬解析的时候，对于使用了绑定变量的字段，不再采用以往默认的 COST 来分析的方式，而是会对绑定变量进行窥探，也就是优化器采集绑定变量的值，用窥探到的绑定变量的值来计算 CBO 的开销，生成执行计划。下面的示例可以说明绑定变量窥探技术的工作原理。

```
SQL> var a number;
SQL> exec :a:=10;
PL/SQL 过程已成功完成。
SQL> select * from dept where deptno=:a;
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, NULL, 'ADVANCED'));
----ADVANCED 是 Oracle 内部使用的未正式发布的参数
PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------
SQL_ID  6a2y9jbwu5fz9, child number 0
-------------------------------------
select * from dept where deptno=:a
Plan hash value: 2852011669
---------------------------------------------------------------------------------
| Id  | Operation                   |Name    | Rows  |Bytes  |Cost (%CPU)  |Time    |
-------------------------------------------------------------------------------
|  0  | SELECT STATEMENT            |        |       |       |    1 (100)  |        |
|  1  | TABLE ACCESS BY INDEX ROWID | DEPT   |   1   |  20   |    1   (0)  |00:00:01|
|* 2  | INDEX UNIQUE SCAN           |PK_DEPT |   1   |       |    0   (0)  |        |
---------------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1 / DEPT@SEL$1
   2 - SEL$1 / DEPT@SEL$1
PLAN_TABLE_OUTPUT
```

```
--------------------------------------------------------------------------------
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('10.2.0.1')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX(@"SEL$1" "DEPT"@"SEL$1" ("DEPT"."DEPTNO"))
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
      END_OUTLINE_DATA
  */
Peeked Binds (identified by position):
--------------------------------------
   1 - :A (NUMBER): 10                   ------从这里可以看出 A:=10 被用来 PARSE 这个 SQL
Predicate Information (identified by operation id):
---------------------------------------------------
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
   2 - access("DEPTNO"=:A)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - "DEPTNO"[NUMBER,22], "DEPT"."DNAME"[VARCHAR2,14],
       "DEPT"."LOC"[VARCHAR2,13]
   2 - "DEPT".ROWID[ROWID,10], "DEPTNO"[NUMBER,22]
已选择 50 行。
SQL> exec :a:=20;   -------看看 A:=20 后会发生什么
PL/SQL 过程已成功完成。
SQL> select * from dept where deptno=:a;
    DEPTNO DNAME          LOC
---------- -------------- -------------
        20 RESEARCH       DALLAS
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, NULL, 'ADVANCED'));
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
SQL_ID  6a2y9jbwu5fz9, child number 0
-------------------------------------
select * from dept where deptno=:a
Plan hash value: 2852011669
----------------------------------------------------------------------------------
| Id  | Operation                   | Name    | Rows |Bytes |Cost (%CPU) | Time    |
----------------------------------------------------------------------------------
|  0  | SELECT STATEMENT            |         |      |      |    1 (100) |         |
|  1  | TABLE ACCESS BY INDEX ROWID | DEPT    |   1  |  20  |    1  (0)  | 00:00:01|
|* 2  | INDEX UNIQUE SCAN           |PK_DEPT  |   1  |      |    0  (0)  |         |
----------------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1 / DEPT@SEL$1
   2 - SEL$1 / DEPT@SEL$1
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('10.2.0.1')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX(@"SEL$1" "DEPT"@"SEL$1" ("DEPT"."DEPTNO"))
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
      END_OUTLINE_DATA
  */

Peeked Binds (identified by position):
--------------------------------------

   1 - :A (NUMBER): 10                  -------peeked binds 的值并没有改变，还是那个值。
Predicate Information (identified by operation id):
---------------------------------------------------
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
   2 - access("DEPTNO"=:A)
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "DEPTNO"[NUMBER,22], "DEPT"."DNAME"[VARCHAR2,14],
       "DEPT"."LOC"[VARCHAR2,13]
   2 - "DEPT".ROWID[ROWID,10], "DEPTNO"[NUMBER,22]
已选择 50 行。
SQL> spool off
```

如果刷新一下共享池，会出现什么结果呢？

```
SQL> ALTER SYSTEM FLUSH SHARED_POOL;                 系统已更改。
SQL> exec :a:=20;                                    PL/SQL 过程已成功完成。
SQL> select * from dept where deptno=:a;
DEPTNO DNAME          LOC
---------- -------------- -------------
20 RESEARCH       DALLAS
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, NULL, 'ADVANCED'));
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
SQL_ID  6a2y9jbwu5fz9, child number 0
-------------------------------------
select * from dept where deptno=:a
Plan hash value: 2852011669
---------------------------------------------------------------------------------
|Id| Operation                   | Name   |Rows | Bytes   |Cost (%CPU) | Time     |
---------------------------------------------------------------------------------
|0 | SELECT STATEMENT            |        |     |         |   1 (100)  |          |
|1 | TABLE ACCESS BY INDEX ROWID|DEPT    |  1  |   20    |   1   (0)  | 00:00:01|
|*2| INDEX UNIQUE SCAN           |PK_DEPT |  1  |         |   0   (0)  |          |
---------------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1 / DEPT@SEL$1
   2 - SEL$1 / DEPT@SEL$1
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('10.2.0.1')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      INDEX(@"SEL$1" "DEPT"@"SEL$1" ("DEPT"."DEPTNO"))
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
      END_OUTLINE_DATA
  */
Peeked Binds (identified by position):
--------------------------------------
   1 - :A (NUMBER): 20
--------------BIND PEEK 的值改变了。说明重新分析的时候会重新做 BIND PEEKING
Predicate Information (identified by operation id):
---------------------------------------------------
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
   2 - access("DEPTNO"=:A)
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "DEPTNO"[NUMBER,22], "DEPT"."DNAME"[VARCHAR2,14],
       "DEPT"."LOC"[VARCHAR2,13]
   2 - "DEPT".ROWID[ROWID,10], "DEPTNO"[NUMBER,22]
已选择 50 行。
```

通过该示例，我们可以看出绑定窥探是如何进行的。当硬解析发生的时候，绑定变量窥探就会进行。一旦理解了这个原理，就不难明白一些以前经常被认为是“灵异事件”的现象。

第一个“灵异现象”是，某条 SQL 一直执行得很正常，突然就变得特别慢了。碰到这类问题如何分析呢？首先我们要看看系统资源是否存在瓶颈，是否存在一些异常的问题，比如出现换页、CPU 资源不足、IO 问题等。这些需要通过操作系统的监控工具进行分析。

如果分析发现系统资源情况是正常的，下一步就需要查看系统的主要等待事件是否正常，是否存在一些特殊的情况。这个可以通过下面的 SQL 来分析：

```
Select count(*),event from v$session_wait group by event order by count(*);
```

如果上面的 SQL 发现了一些异常，那么就要具体分析这些异常，看看这些等待和什么相关。这个分析过程不是本节讨论的重点，因此我们不再深入探讨。如果从等待事件上没有发现什么特

殊的情况或者发现可能是某些SQL性能不佳导致，那么就需要进一步分析SQL的问题了。接下来，我们可以检查SQL的执行计划，看看执行计划是否合理。对于一些较为简单的SQL，可以很容易地发现执行计划中可能存在的问题，而如果执行计划十分复杂，有十多行甚至几十行，那么分析执行计划的最佳方法就是保存一部分这些SQL的基线，为这些关键SQL的正常执行计划建立一个档案库。通过和历史数据的比对来发现问题是最准确和快捷的。

经过上面的分析，我们可能会发现某个使用绑定变量的SQL的执行计划突然发生了变化，变化后的执行计划明显是不合理的，正是这个原因导致了故障的发生。既然问题已经找到了，比较喜欢较真的朋友可能会有些疑问，为什么正常的SQL的执行计划就发生了改变呢？如果排除了相关表的数据发生较大变化的可能性，那么执行计划改变的主要原因就可能是绑定变量窥探。由于绑定变量窥探只在硬解析发生的时候进行，根据这一点就可以推测可能出现的场景。由于共享池空闲空间比较少，所以部分共享池的可重用的对象被刷出了，其中正好包含了这条SQL的一部分。因此在下一次执行该SQL的时候，就出现了SQL突然变慢的现象。这时绑定变量的值比较特殊，从而导致了生成的执行计划和以往的不同。如果再深入分析这条SQL，我们就会发现，绑定变量对应的字段的数据不是均匀分布的，而是倾斜的，某些取值的数据量比较大，而另一些取值的数据量又比较小。

另外一个“灵异现象”是，在RAC的不同节点上，相同的两个SQL的执行计划不同。如果发现这条SQL中使用了绑定变量，并且绑定变量对应的字段的值域也是倾斜的，那么绑定变量窥探导致这个问题的可能性就十分高了。这是因为RAC的多个实例维护自己独立的共享池，一条SQL在多个实例上执行，必须分别解析，在RAC环境下，SQL的执行计划是实例范围内的，不会跨实例。如果两个实例在解析这条SQL的时候，窥探到的绑定变量的值是不同的，那么就可能导致生成的执行计划也不同。

碰到这样的问题如何解决呢？可能大家首先想到的是刷新共享池，确实，刷新共享池可能将某个SQL的游标刷出，不过如果这个游标被锁住了，那么刷新共享池是无效的。而且在业务高峰期进行此操作也存在一定的风险，共享池有可能会被挂起，因此不建议采用这种方式。要解决好这个问题，就需要了解什么情况会导致SQL被重新解析。已知的情形很多，比如，相关的表和索引出现了变更，相关的表的权限发生了变更，表和索引的统计数据发生了变更等，都能够导致某条SQL被重新硬解析。因此我们一般采用对相关表授权的方式来让SQL重新解析。还有一个更为安全的方法，就是先将该表的统计数据导出，然后再导入，这样表的统计数据就会发生变化，从而导致游标被重新硬解析。

正是由于绑定变量窥探的局限性，Oracle 11*g*推出了一种新的游标共享技术ACS（Adaptive Cursor Sharing，自适应游标共享），使绑定变量的窥探技术有所改变。每次绑定变量都会被窥视，不过并不是每次窥视都会产生新的执行计划，一旦发现某个绑定变量取值和已有的值域有所不同，就会产生新的子游标，生成新的执行计划。使用ACS，上述的类似问题将会得到解决。不过目前ACS技术尚不完善，会出现一些Bug，导致某些游标的子游标数量十分大，从而影响共享池和SQL执行的性能，在使用时需要注意。

3.2.4　OPEN CURSOR 和 OPEN_CURSORS 参数

OPEN_CURSORS 是一个十分有趣的参数，经常有 DBA 会发现自己系统中的打开游标数非常大。我们来看一个例子。

```
SQL>select sid,value from v$sesstat a,v$statname b where a.statistic#=b.statistic# and
name='opened cursors current' order by 2;
       SID      VALUE
---------- ----------
      5430         93
      3527         95
      4055         96
      4090         97
      2012         98
      1819         98
      5349        102
      1684        103
      1741        116
      4308        169
      1970        170
      1369        181
      4208        184
       887        214
      5215        214
      3518        214
       868        214
      1770        215
      4050        215
      1809        231
      3010        235
       762        237
       731        471
      4013       1066
      2648       1152
      2255       1172
      2322       2620
```

这个系统的 OPEN_CURSORS 参数设置为 3000，而会话中当期打开游标最大的会话居然达到了 2620。在一般人的眼里，游标使用后就关闭了，打开的游标的数量应该不会太多。难道应用程序出现了游标泄漏，有些应用使用了游标没有关闭？实际上，我们对打开游标的概念一直存在误解，认为只有正在获取的游标是打开状态，而一旦获取结束，关闭游标后，游标就处于关闭状态了，因此一个会话中打开状态的游标数量应该很少。事实上不是这样的。某些游标在程序中已经关闭了，但是 Oracle 为了提高游标的性能，会对其进行缓冲，这些缓冲的游标在程序中的关闭只是一种软关闭，事实上，在会话中并未关闭，而是存放在一个游标缓冲区中。

在 Oracle 9.2.0.5 之前，OPEN_CURSORS 参数的作用是双重的，一方面是限制一个会话打开游标的总量，另外一方面，OPEN_CURSORS 参数也作为 PL/SQL CURSOR 的缓冲。在 PL/SQL 中，如果某个游标关闭了，它不会马上硬关闭，而是首先保存在游标缓冲中。如果这个会话当前

打开的游标数量还没有达到 OPEN_CURSORS 参数的值，那么就可以先保持打开状态。如果当前打开的游标数量已经达到了 OPEN_CURSORS 参数的限制，那么首先会关闭一个被缓冲的、实际当时并未打开的游标。如果缓冲池中的所有游标都是实际打开的，那么就会报错 ORA-1000：maximum open cursors exceeded。

Oracle 9.2.0.5 以后，OPEN_CURSORS 参数不再承担 PL/SQL 缓冲的工作，PL/SQL 中的 SQL 也可以使用 SESSION_CACHED_CURSORS 的会话缓冲了。这个参数就成为了一个纯粹的限制。

虽然如此，OPEN_CURSORS 参数仍然和游标的缓冲机制密切相关，因为这个参数限制了当前某个会话打开游标的最大值。设置一个较大的 OPEN_CURSORS 参数，可以避免出现 ORA-1000 错误，同时也可以让会话缓冲更多的游标，改善 SQL 解析的性能。不过将这个参数设置得较大会占用较多的 PGA 空间，消耗一定的物理内存。因此它并不是设置得越大越好，一般的 OLTP 系统中，1000～3000 就足够了。在共享服务器模式的系统中，该参数的设置要略微保守一些，因为这个参数越大，占用的 SGA 空间也就越大。

另外要注意的是，从 Oracle 9.0 开始，这个参数就已经是动态的了，可以随时动态调整。

3.2.5　CURSOR_SPACE_FOR_TIME 参数

CURSOR_SPACE_FOR_TIME 是一个十分“霸道”的参数，自从 Oracle 8*i* 引入这个参数以来，就一直饱受争议。CURSOR_SPACE_FOR_TIME 的引入目的是减少游标锁的数量，从而减少共享池相关闩锁的争用。

“CURSOR_SPACE_FOR_TIME=TRUE”的原理是，当父游标被开启的时候，所有的子游标及其关联的相关对象全部被锁住，从而确保该游标所有的相关信息都是一致的，而且是完全保存在共享池中的，进而提高游标执行时的效率。如果这个参数没有被设置，那么只有当 SQL 处于执行阶段时，相关的对象才会被锁住。一旦执行阶段完成，就没必要再锁住整个游标了。在上一节讨论 OPEN_CURSORS 参数的时候，我们知道很多游标执行完毕后并不是物理关闭的，而是放在游标高速缓存里，那么这些游标就会被锁住，无法被释放继续使用。直到会话退出，才会真正关闭这些游标，而直到最后一个使用游标的会话退出，该游标才能被换出。这样就可能导致大量的共享池对象被锁住，使共享池碎片化，严重时会导致共享池无法分配足够大小的连续空间，系统将出现故障，甚至宕机。

虽然这个参数能够减少共享池的闩锁争用，但是它也有很大的副作用，就是会增加共享池碎片的机会。因为只要某个游标的父游标没有关闭，那么该游标所有的相关对象都会被锁在共享池中，不会被换出，这样就加大了共享池碎片化的趋势。如果存在大量这样的游标，那么共享池就会变得十分零碎，出现 ORA-4031 错误，宕机的可能性就很大了。

正是因为上述原因，使用 CURSOR_SPACE_FOR_TIME 要十分谨慎，一般不建议使用。如果确实需要使用该参数，就要确保共享池是较为充足的，并且设置该参数后共享池碎片化趋势不是十分明显。如果使用了这个参数，建议定期重启数据库实例，从而缓解共享池碎片化的

趋势。

从 10.2.0.5 版本开始，库缓存锁已经被 mutex 全面替代，因此也就不再需要这个参数了。CURSOR_SPACE_FOR_TIME 完成了其使命，被彻底废除了。

3.2.6 SESSION_CACHED_CURSORS 参数和 OPEN_CURSORS

SESSION_CACHED_CURSORS 参数的引入是个偶然。Oracle FORMS 的开发人员发现业务逻辑要求表单之间需要经常来回切换，而一旦某个表单切换到另外一个表单，旧表单上的所有游标就必须全部关闭，下次切换回来时，又需要再次打开，这样就大大降低了表单切换的效率。人们开始考虑能否开发一种类似于软关闭的方式，使游标不被真正关闭，从而提高表单应用的效率。于是 SESSION_CACHED_CURSORS 参数就应运而生了。在 UGA 中建立一个独立的游标缓冲池，将常用的游标缓冲起来，下次执行时可以直接从缓冲池中取出游标，不需要再次解析。这个缓冲池和 OPEN_CURSORS 所控制的游标缓冲池是不同的，前者最初是为 PL/SQL 中的 CURSOR 服务，而 SESSION_CACHED_CURSORS 的缓冲在 9.2.0.5 版本之前是不能缓冲 PL/SQL 中的游标的，9.2.0.5 版本以后才开始支持。游标被放入会话缓冲池中的前提是被多次解析，在一个游标被解析 3 次后，就会被放入会话缓冲区中。

SESSION_CACHED_CURSORS 参数在共享池优化工作中是十分常用的。这个参数的主要功能是将某个会话中常用的 SQL 放入 UGA 中的会话缓冲区里，以便于下次调用，这样就不需要再做解析，从而减少了共享池的争用。

如果使用绑定变量的目的是减少硬解析，从而改善共享池，提高并发能力，那么使用 SESSION_CACHED_CURSORS 参数的目的就是减少软解析，从而进一步提高共享池的性能。对于某些系统而言，不仅仅是硬解析需要优化，大量的并发执行，使软解析也可能成为系统的瓶颈，相关示例的 AWR 报告如下。

```
Load Profile
~~~~~~~~~~~~                      Per Second        Per Transaction
                               ---------------      ---------------
Redo size:                        1,989,039.62         12,816.67
Logical reads:                      181,739.18          1,171.06
Block changes:                        7,007.77             45.16
Physical reads:                       2,112.79             13.61
Physical writes:                          0.57              0.00
User calls:                          12,191.86             78.56
Parses:                               8,958.80             57.73
Hard parses:                             49.07              0.32
Sorts:                                  210.70              1.36
Logons:                                   0.03              0.00
Executes:                            11,725.29             75.55
Transactions:                           155.19
```

从报告中可以看出，平均每秒执行的数量为 11 725，每秒的解析数量为 8958。在这种情况下，共享池的争用十分严重，相关数据如下：

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:    99.96       Redo NoWait %:    100.00
            Buffer  Hit   %:    98.84    In-memory Sort %:    100.00
            Library Hit   %:   107.03        Soft Parse %:     99.45
         Execute to Parse %:    23.59         Latch Hit %:     95.60
Parse CPU to Parse Elapsd %:    2.89     % Non-Parse CPU:     94.08

 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:    9.52    9.57
    % SQL with executions>1:   94.13   94.34
  % Memory for SQL w/exec>1:   89.00   92.95

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                       % Total
Event                               Waits    Time (s)  DB Time   Wait Class
--------------------------- ---------- ----------- -------- -----------
latch: library cache          179,862       3,360    35.73    Concurrency
CPU time                                    1,391    14.79
latch: cache buffers chains    21,376         642     6.83    Concurrency
latch: library cache lock      26,875         561     5.96    Concurrency
latch: library cache pin       30,675         519     5.52    Concurrency
```

我们可以看到，虽然共享池的命中率很高，但是库缓存闩锁的争用十分严重。相关等待接近总体的 50%。在这种极端的情况下，通过加大 SESSION_CACHED_CURSORS 参数，可以有效地减少软解析的数量，从而缓解共享池相关闩锁的争用。

针对这个案例，建议采取两项措施，第一项是设置 SESSION_CACHED_CURSORS 为 200。参数调整后并发量会大幅度提升，并发量增加后 DB Cache 的压力就会有所增加，从而影响整体的优化效果。为了保障性能，应同时采取第二项措施，将 DB Cache 加大 5 ~ 10GB。

事实证明加大 SESSION_CACHED_CURSORS 参数的效果十分明显，共享池闩锁争用明显降低，每秒平均事务数量从 155 提高到 300 左右。

为什么调整 SESSION_CACHED_CURSORS 参数会有这么大的效果呢？为了了解这一点，我们来做一个实验，分析一下 SESSION_CACHED_CURSORS 参数对 library cache pin 和 library cache lock 的影响。

在分析 library cache pin/lock 前，首先需要了解 10049 事件。从 10.2 版本开始，10049 事件可以全面监控库缓存的 PIN/LOCK 和 INVALIDATION，其参数在 9*i* 和 10*g* 版本中有所不同。

要使用 10049 事件，首先需要找出 SQL 的散列值，用散列值的低位作为 LEVEL 的高位，加上 0X2030（TRACE PIN/LOCK），产生 LEVEL 的值。

```
var id number;
exec :id:=1099;
select empno,ename from emp where empno=:id;
select hash_value from v$sqlarea where sql_text like 'select empno,ename%';
HASH_VALUE
----------
2892642740
用计算器看看十六进制：
AC6A39B4
```

这个十六进制数的低位是 39B4，我们要 TRACE PIN/LOCK，因此 TRACE LEVEL 为 39B42030，转换为十进制就是 968106032。下面首先将 SESSION_CACHED_CURSORS 设置为 0，关闭会话的 CURSOR CACHE，看看会发生什么情况。

```
alter session set session_cached_cursors=0 ;
alter session set events '10049 trace name context forever,level 968106032';
```

该设置对相关库缓存的 PIN 和 LOCK 进行跟踪操作。准备结束，通过多次执行下面的语句来检查 library cache pin 和 library cache lock 的情况。

```
exec :id:=1010;
select empno,ename from emp where empno=:id;
exec :id:=1011;
select empno,ename from emp where empno=:id;
```

可以看到，每次执行，跟踪文件中的 PIN 和 LOCK 就会增加。

```
KGLTRCLCK kglget     hd = 0x1FBCB8A4  KGL Lock addr = 0x20F75530 mode = N
KGLTRCLCK kglget     hd = 0x1FBCB7C0  KGL Lock addr = 0x20FA8970 mode = N
KGLTRCPIN kglpin     hd = 0x1FBCB7C0  KGL Pin  addr = 0x20F259D0 mode = S
KGLTRCPIN kglpndl    hd = 0x1FBCB7C0  KGL Pin  addr = 0x20F259D0 mode = S
KGLTRCLCK kgllkdl    hd = 0x1FBCB7C0  KGL Lock addr = 0x20FA8970 mode = N
KGLTRCLCK kgllkdl    hd = 0x1FBCB8A4  KGL Lock addr = 0x20F75530 mode = N
```

首先来看这里涉及的两个地址，library_cache 的跟踪信息如下：

```
Bucket 14772:
  LIBRARY OBJECT HANDLE: handle=1fbcb8a4 mutex=1FBCB958(1)
  name=select empno,ename from emp where empno=:id
  hash=cedcee1ccc9795f332f72482ac6a39b4 timestamp=12-24-2007 21:55:26
  namespace=CRSR flags=RON/KGHP/TIM/PN0/SML/KST/DBN/MTX/[120100d0]
  kkkk-dddd-llll=0000-0001-0001 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
  lwt=1FBCB900[1FBCB900,1FBCB900] ltm=1FBCB908[1FBCB908,1FBCB908]
  pwt=1FBCB8E4[1FBCB8E4,1FBCB8E4] ptm=1FBCB8EC[1FBCB8EC,1FBCB8EC]
  ref=1FBCB920[1FBCB920,1FBCB920] lnd=1FBCB92C[1FBCB6F0,23192730]
    LIBRARY OBJECT: object=2052e5e4
    type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
    CHILDREN: size=16
    child#    table reference   handle
    ------ -------- --------- --------
         0 2052d080  2052cd34 1fbcb7c0
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 23169fcc 2052e67c I/-/A/-/-    0 NONE   00
  Bucket 14772 total object count=1
```

上面的信息就是这个 SQL 的父游标的句柄，而 1FBCB7C0 是其子游标句柄的地址。

```
  LIBRARY OBJECT HANDLE: handle=1fbcb7c0 mutex=1FBCB874(0)
  namespace=CRSR flags=RON/KGHP/PN0/[10010000]
  kkkk-dddd-llll=0000-0000-0000 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
  lwt=1FBCB81C[1FBCB81C,1FBCB81C] ltm=1FBCB824[1FBCB824,1FBCB824]
  pwt=1FBCB800[1FBCB800,1FBCB800] ptm=1FBCB808[1FBCB808,1FBCB808]
  ref=1FBCB83C[2052CD34,2052CD34] lnd=1FBCB848[1FBCB848,1FBCB848]
```

```
    CHILD REFERENCES:
    reference latch flags
    --------- ----- -------------------
     2052cd34     0 CHL[02]
    LIBRARY OBJECT last freed from HPD addn data CBK
```

从上面的跟踪信息可知，执行 SQL 过程中，需要 4 个 LOCK，两个 PIN。那么如果是被缓存了的游标会怎么样呢？

```
22:01:01 SQL> alter session set session_cached_cursors=20;
会话已更改。
22:14:35 SQL> select empno,ename from emp where empno=:id;
未选定行
22:14:39 SQL> select empno,ename from emp where empno=:id;
未选定行
22:14:42 SQL> select empno,ename from emp where empno=:id;
未选定行
22:14:56 SQL> select empno,ename from emp where empno=:id;
未选定行
22:16:09 SQL> select empno,ename from emp where empno=:id;
```

相关的跟踪信息如下：

```
*** 2007-12-24 22:16:21.722
KGLTRCPIN kglpin     hd = 0x1FBBF998  KGL Pin  addr = 0x21248078 mode = S
KGLTRCPIN kglpndl    hd = 0x1FBBF998  KGL Pin  addr = 0x21248078 mode = S
```

令人惊奇的是，library cache lock 不见了，对父游标的 LOCK/PIN 都消失了，这就是 SESSION_CACHED_CURSORS 优化分析过程的一个直观表现。为什么不需要对子游标和父游标施加 library cache lock 呢？我们先来看父游标句柄：

```
Bucket 14772:
  LIBRARY OBJECT HANDLE: handle=1fbcb8a4 mutex=1FBCB958(1)
  name=select empno,ename from emp where empno=:id
  hash=cedcee1ccc9795f332f72482ac6a39b4 timestamp=12-24-2007 22:14:39
  namespace=CRSR flags=RON/KGHP/TIM/KEP/PN0/SML/KST/DBN/MTX/[120100d4]
  kkkk-dddd-llll=0001-0001-0001 lock=N pin=0 latch#=3 hpc=0002 hlc=0002
  lwt=1FBCB900[1FBCB900,1FBCB900] ltm=1FBCB908[1FBCB908,1FBCB908]
  pwt=1FBCB8E4[1FBCB8E4,1FBCB8E4] ptm=1FBCB8EC[1FBCB8EC,1FBCB8EC]
  ref=1FBCB920[1FBCB920,1FBCB920] lnd=1FBCB92C[2329CBD8,231B6A88]
    DEPENDENCY REFERENCES:
    reference latch flags
    --------- ----- -------------------
     20553df0     0 [60]
    LOCK OWNERS:
        lock     user  session count mode flags
    -------- -------- -------- ----- ---- ------------------------
    212383e8 2372d86c 2372d86c     1 N    [00]
    LIBRARY OBJECT: object=2059dea0
    type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
    CHILDREN: size=16
    child#    table reference   handle
    ------ -------- --------- --------
         0 205544a0  20554154 1fbbf998
```

```
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 232307b8 2059df38 I/P/A/-/-    0 NONE   00
  Bucket 14772 total object count=1
```

可以看出，和普通的库缓存不同，lock 不是 0，而是 N，这说明在该对象上已经有一个空锁，因此不需要再加锁了。接下来再看看子游标。


```
 LIBRARY OBJECT HANDLE: handle=1fbbf998 mutex=1FBBFA4C(0)
  namespace=CRSR flags=RON/KGHP/PN0/[10010000]
  kkkk-dddd-llll=0000-0041-0041 lock=N pin=0 latch#=3 hpc=0002 hlc=0002
  lwt=1FBBF9F4[1FBBF9F4,1FBBF9F4] ltm=1FBBF9FC[1FBBF9FC,1FBBF9FC]
  pwt=1FBBF9D8[1FBBF9D8,1FBBF9D8] ptm=1FBBF9E0[1FBBF9E0,1FBBF9E0]
  ref=1FBBFA14[20554154,20554154] lnd=1FBBFA20[1FBBFA20,1FBBFA20]
    CHILD REFERENCES:
    reference latch flags
    --------- ----- -------------------
     20554154     0 CHL[02]
    LOCK OWNERS:
        lock     user  session count mode flags
    -------- -------- -------- ----- ---- ------------------------
    20f89814 2372d86c 2372d86c     1 N    [00]
    LIBRARY OBJECT: object=20553cb4
    type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
    DEPENDENCIES: count=1 size=16
    dependency#    table reference   handle position flags
    ----------- -------- --------- -------- -------- -------------------
              0 2046173c  20461484 1fbcb668       24 DEP[01]
    READ ONLY DEPENDENCIES: count=1 size=16
    dependency#    table reference   handle flags
    ----------- -------- --------- -------- -------------------
              0 20554070  20553df0 1fbcb8a4 /ROD/KPP[60]
    AUTHORIZATIONS: count=1 size=16 minimum entrysize=16
    00000000 36000000 00020000 00000000
    ACCESSES: count=1 size=16
    dependency# types
    ----------- -----
              0 0009
    TRANSLATIONS: count=1 size=16
    original    final
    -------- --------
    1fbcb668 1fbcb668
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 23217f84 20553e04 I/P/A/-/-    0  NONE   00
        6 20461370 1fcf1d3c I/-/A/-/-    0  NONE   00
```

子游标上同样有一个空锁。由于被缓存了的游标不需要再进行 PIN 操作，这样就减少了共享池的闩锁争用，有效地提高了 SQL 并发执行的效率。这就很好地解释本节开始时那个优化案例的效果。

SESSION_CACHED_CURSORS 可以有效地减少软分析，那么在什么情况下需要减少软分析呢？我们看看代码清单 3-5 所示的查询。

代码清单 3-5

```
SQL> col cursor_cache_hits format a20 truncate;
SQL> col soft_parses format a20 truncate;
SQL> col hard_parses format a20 truncate;
SQL> select
  to_char(100 * sess / calls, '9999990.00') || '%' cursor_cache_hits,
  to_char(100 * (calls - sess - hard) / calls, '999990.00') || '%' soft_parses,
  to_char(100 * hard / calls, '999990.00') || '%' hard_parses
from
  ( select value calls from v$sysstat where name = 'parse count (total)' ),
  ( select value hard  from v$sysstat where name = 'parse count (hard)' ),
  ( select value sess  from v$sysstat where name = 'session cursor cache hits' );
  2    3    4    5    6    7    8
CURSOR_CACHE_HITS       SOFT_PARSES          HARD_PARSES
-------------------- -------------------- --------------------
      85.34%              14.61%                0.06%
```

从上面的查询结果来看，游标缓存的命中率是 85.34，软解析的比例是 14.16%，硬解析的比例是 0.06%。目前这个系统的 SESSION_CACHED_CURSORS 参数设置为 200。加大 SESSION_CACHED_CURSORS 参数能否提高命中率，进一步减少软解析呢？我们可以用代码清单 3-6 所示的查询继续分析。

代码清单 3-6

```
Select  'session_cached_cursors'  parameter, lpad(value, 5)  value,
decode(value, 0, '  n/a', to_char(100 * used / value, '990') || '%')  usage
from
  ( select
      max(s.value)  used
     from
      v$statname  n,
      v$sesstat  s
     where
      n.name = 'session cursor cache count' and
      s.statistic# = n.statistic#
  ),
 ( select
        value
     from
      v$parameter
     where
      name = 'session_cached_cursors'
  ) ;

PARAMETER                  VALUE      USAGE
---------------------- ---------- ----------
session_cached_cursors     200        100%
```

从查询结果来看，CACHE 的使用率是 100%。这个 SQL 实际上是通过会话的信息查找会话缓冲使用的最大值，因此查询结果是 100%并不一定就说明 SESSION_CACHED_CURSORS 参数太小了，一定要参考上面的软解析百分比，综合分析。如果有必要的话，加大 SESSION_CACHED_CURSORS 参数，还可以进一步减少软解析。一般来说，在一个充分调优的 OLTP 系统中，SESSION CACHE 的命中率可以达到 90%甚至 95%以上。

3.2.7　CURSOR_SHARING 和游标共享

谈到共享池，就不得不提及 CURSOR_SHARING 参数。有些应用可能写得不够好，里面使用了大量的非绑定变量，针对这种情况，Oracle 从 8.0 版本开始支持通过设置 CURSOR_SHARING 参数来共享没有使用绑定变量的类似 SQL。刚开始的时候，Oracle 只是提供了 FORCE 设置，强制性地共享类似 SQL，从而解决了未使用绑定变量导致 SQL 无法共享的问题。从 9*i* 开始，CURSOR_SHARING 能够支持 SIMILAR 参数值。SIMILAR 选项的出现是为了弥补 FORCE 设置的不足。在 8.1.6 和 8.1.7 这两个版本中，如果使用了 CURSOR_SHARING=FORCE，那么绑定窥探将不再进行，这样就导致了部分应用在使用 CURSOR_SHARING 参数后，执行计划出现偏差，使得大量关键 SQL 采用了较差的执行计划，导致系统的总体性能没有提升反而下降。在 9*i* 版本中使用 SIMILAR，对于存在柱状图的列，可以进行绑定窥探，从而解决了 FORCE 产生的问题，这样既可以共享 SQL，又不会产生较差的执行计划。

似乎一切都很完美，不过由于实际应用的复杂性，SIMILAR 又引入了新的问题。对于 SIMILAR 如何判断 SQL 是否可以共享的资料十分有限，因此老白也没有对此进行过十分深入的研究，只是根据 Oracle Metalink 的文档了解到，使用了类似<>、between、like 和!=这些判断条件的 SQL，在 CURSOR_SHARING=SIMILAR 时是无法共享的。这将导致某条 SQL 由于 CURSOR_SHARING = SIMILAR 而被替换为文本，比如：

```
SELECT * FROM TAB1 WHERE COL1>10;
SELECT * FROM TAB1 WHERE COL1>20;
SELECT * FROM TAB1 WHERE COL1>30;
```

这三条 SQL 被认为是同一条 SQL：

```
SELECT * FROM TAB1 WHERE COL1:"SYS_B_0";
```

通过下面的示例可以了解到不同的判断条件对 CURSOR_SHARING=SIMILAR 情况下 SQL 共享的影响：

```
alter session set cursor_sharing=similar;
select count(*) from t1 where object_id>10;
select count(*) from t1 where object_id>20;
select count(*) from t1 where object_id>30;
select count(*) from t1 where object_id=10;
select count(*) from t1 where object_id=20;
select count(*) from t1 where object_id=30;

col sql_text format a60 trunc
```

```
set line 132
select sql_text,version_count  from v$sqlarea
    where sql_text like '%from t1%';
SQL_TEXT                                                      VERSION_COUNT
------------------------------------------------------------ -------------
select count(*) from t1 where object_id>:"SYS_B_0"                        3
select count(*) from t1 where object_id=:"SYS_B_0"                        1
```

从上面的测试可以看出，“=”条件的 SQL 被很好地共享了，而“>”条件的 SQL 无法共享，另外，在 V$SQL_SHARED_CURSOR 中，我们无法看到任何不能共享的原因。

可能还有一些朋友没有看出其中的问题，SQL 无法共享又能怎么样呢？实际上，在生产环境中，如果存在大量这样的 SQL，那么这些 SQL 就会由于 CURSOR_SHARING 参数的设置而被合并为一条 SQL，从而拥有相同的父游标，不过由于最终游标不能共享，因此在父游标下会产生大量的子游标（几百甚至上千）。在访问这些游标时，由于子游标的数量巨大，查找相关子游标的时间就会很长。期间可能导致大量库缓存或者 MUTEX 的闩锁争用。上千个无法共享的游标共享一个父游标，其效率可能还不如让每个游标都拥有独立的父游标。虽然这会占用更多的共享池空间，但是如果共享池足够大，其访问过程中的库缓存和 MUTEX 相关闩锁等待都会少很多。

实际上，Oracle 11*g* 提供了更为优秀的游标共享算法，既避免了类似 CURSOR_SHARING=FORCE 的执行计划不准确的问题，又避免了 CURSOR_SHARING=SIMILAR 的大量游标不能共享的问题。这种解决方案就是 11*g* 版本中的 ACS。在 ACS 启用的情况下，绑定窥探技术得到了发挥，在判断游标是否可以共享的时候，会根据绑定变量的值进行分析。如果值域对应的选择性可以使用原有的执行计划，就共享 SQL，否则就创建一个子游标。这种技术既解决了游标共享的问题，又有效地避免了因绑定变量值不同而采取不同执行计划的问题，可以算是一种两全其美的方法。

如果使用 ACS，那么对于没有使用绑定变量的应用，我们只需要使用 CURSOR_SHARING=FORCE，其他的事情就交给 ACS 去完成了。实际上，在 11*g* 的早期版本中，ACS 的 Bug 比较多，还是有可能出现产生大量子游标的情况，使用者要十分小心。如果发现这个问题，应马上关闭 ACS，这时我们可以通过设置 _OPTIMIZER_EXTENDED_CURSOR_ SHARING_REL=NONE 来关闭 ACS 的功能。相关 Bug 可能导致的问题如表 3-2 所示。

表　3-2

Bug NO	问题描述	影响版本
10182051	有多个绑定变量的SQL，在V$SQL中的IS_SHARBLE=Y，但是有大量不同的版本	11.2.0.2（11.2.0.3修复）
7213010	可在V$SQL_CS_SELECTIVITY中看到一个CURSOR下存在大量的equivalent / overlapping	11.1.0.6（11.1.0.7修复）
6644714	可在V$SQL_SHARED_CURSOR中看到大量CHILD CURSOR的LOAD_OPTIMIZER_STATS='Y'	11.1.0.6（11.1.0.7修复）

此外，如果我们在 11*g* 版本的数据库中启用了 ACS，并使用了 CURSOR_SHARING=SIMILAR，可能会导致十分严重的问题——大量的游标被合并了，但是不能共享，一个父游标下存在大量不可共享的子游标。因此，在 11*g* 版本中，如果开启了 ACS 功能，那么就不建议使用 CURSOR_

SHARING=SIMILAR。在 Oracle 12C 版本中，CURSOR_SHARING=SIMILAR 已被彻底禁止（不再支持这个设置）。

3.2.8　游标的关闭

可能很多 DBA 都会碰到这样一件十分奇怪的事情，就是某个会话中打开的游标数量会十分庞大，比如，在某个生产系统中进行这样的查询：

```
select sid,b.name,a.value from v$sesstat a,v$statname b where
a.statistic#=b.statistic# and b.statistic#=3 order by 3
       SID NAME                                          VALUE
---------- ---------------------------------------- ----------
      3008 opened cursors current                          214
      2245 opened cursors current                          214
      5314 opened cursors current                          215
      3233 opened cursors current                          215
      2168 opened cursors current                          230
      4535 opened cursors current                          233
      5222 opened cursors current                          234
      4394 opened cursors current                          294
       876 opened cursors current                          310
      1265 opened cursors current                          325
      2320 opened cursors current                          326
      1783 opened cursors current                          331
       734 opened cursors current                          343
       987 opened cursors current                          344
      2241 opened cursors current                          357
       988 opened cursors current                          370
      1055 opened cursors current                          382
      5169 opened cursors current                          393
       302 opened cursors current                          411
      1276 opened cursors current                          422
       797 opened cursors current                          433
      1649 opened cursors current                          638
      4049 opened cursors current                          650
      3853 opened cursors current                          664
      1253 opened cursors current                          692
       370 opened cursors current                          822
      2664 opened cursors current                          834
      4145 opened cursors current                          925
      2322 opened cursors current                         2622
```

为了简洁，老白只截取了部分查询结果。从这些结果上看，会话中打开游标的数量都超过了 200 个，最多的甚至高达 2622 个。一般的应用程序都会在游标使用完毕后将其关闭，怎么会产生这么多打开的游标呢？难道我们碰到了游标泄漏？可是如果出现了游标泄漏，为什么应用没有报错，而系统还能够正常运行呢？要回答这些问题，就需要了解游标关闭的相关知识。

当一个游标使用完毕后，程序就会将这个游标关闭。不过由于前面几节介绍的一些缓冲技术的存在，游标关闭分为软关闭和硬关闭两种。软关闭的游标虽然被客户端程序关闭了，但是实际上在游标缓冲中，仍处于打开状态。这就是我们经常能够发现的会话中存在大量打开状态游标的

主要原因。实际上，如果系统没有出现游标溢出，存在大量软关闭的游标不会对系统产生太大的影响，Oracle 会自动维护这个游标缓冲区，使之较为高效地运作，同时不会因为打开的游标数量太多而导致正常使用的游标无法打开。

只有当游标被硬关闭了，它和会话之间的联系才会彻底断掉。将来这个会话要再次执行同一 SQL 时，就需要重新创建游标的相关数据结构了。

3.2.9　互斥锁和游标

当我们还纠结于 library cache pin 这类等待事件的时候，突然发现，10.2 的系统中 library cache pin 等相关闩的等待比 9*i* 版本要少得多，而带有互斥锁机制的等待事件出现了，且共享池的问题往往也体现在这些事件上。比如，在一个 10.2.0.3 的 AWR 报告中，我们会看到：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~
Event                         Waits    Time(s)   Avg wait(ms)   % DB Time   Wait Class
------------------------- -------- --------- -------------- ---------- -----------
latch: row cache objects      942     13,072        13877       49.73   Concurrency
library cache: mutex X      3,179      2,382          749        9.06   Concurrency
latch: shared pool          2,980      1,559          523        5.93   Concurrency
DB CPU                                 1,101                     4.19
log file sync              27,402        322           12        1.23        Commit
```

这里的 library cache: mutex X 等待出现在 TOP5 EVENT 里。另外，下面的信息来自于 10.2.0.4 的 AWR 报告：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~
Event                            Waits     Time(s) Avg wait(ms)  % Total    Wait Class
                                                                Call Time
------------------------ ---------- --------- ------------- --------- ------------
cursor: pin S wait on X   18,568,538  181,950        10         209.2     Concurrency
library cache load lock       10,314   29,581     2,868          34.0     Concurrency
latch: library cache         134,719   26,756       199          30.8     Concurrency
RDBMS ipc reply               12,258   23,018     1,878          26.5          Other
latch: row cache objects     203,029   19,527        96          22.5     Concurrency
```

其中排在第一位的是 cursor: pin S wait on X，这些新出现的等待事件是什么含义呢？要想解释它们，我们必须首先了解一个从 Oracle 10*g* 才开始引入的新的串行机制——互斥锁机制（MUTEX）。做过多进程或者多线程编程的朋友可能都了解 UNIX 上的信号灯机制。信号灯是 UNIX 系统中早期使用最多的互斥锁机制，而这种机制是一种重量级的锁机制，其成本较高，因此在多线程编程时往往会采用更为轻量级的锁机制——互斥锁机制。互斥锁机制是一种轻量级的同步锁机制，通过维护一个计数器和一个等待队列来实现互斥。

Oracle 的互斥锁和操作系统层面的互斥锁具有一定的相似性，不过并不是相同的机制。Oracle 的互斥锁是由 KGX 层面提供的，和闩类似，它也是采用操作系统的原子操作，以 Compare And Swap（CAS）的方式来自旋获取相关的互斥锁。由于目前的绝大多数 CPU 都已经提供了这种指

令集，因此互斥锁机制可以利用硬件的特性，获得很好的执行效果。不过对于一些不支持 CAS 指令集的 CPU 来说，互斥锁的成本会大大增加，这也是在一些 HP PA-RISC 系统上运行的 10*g* R2 系统经常会出现 cursor: pin S wait X 等待事件的主要原因。

Oracle 互斥锁的结构十分简单，代码清单 3-7 是老白所理解的 KGX 互斥锁的结构说明。

代码清单 3-7

```
struct KGX_MUTEX {
  long                 kgx_mutex;
  ub4                  kgx_mutex_gets;
  ub4                  kgx_mutex_sleep;
  ub4                  kgx_mutex_idn（mutex 唯一性标识）;
}
```

从前文中我们了解了用互斥锁替代 library cache pin，其执行效率会有较大的提高。另外一个好处是，由于互斥锁的使用方法和闩不同，它可以嵌入到其他的结构中使用，比如嵌入到 KGH 结构中。针对 CURSOR STAT，我们来看一个例子。

```
SQL> oradebug setmypid
Statement processed.
SQL> oradebug dump cursor_stats 1;
Statement processed.
SQL>
  Bucket 1 Mutex 0x660cd02c(0,0) tc 1 fc 0
    PARENT 0x65c65458 sqlid=0x7bf48001 phd = 0x68307228
      CHILD 0x6560e4d0 planhsh=0 cnum=0 flags=0x0:
      Parse Count=36
      Disk Reads=0
      Disk Writes=0
      Buffer Gets=2920
      Rows Processed=538
      Serializable Aborts=0
      Fetches=0
      Execution Count=538
      PX Server Execution Count=0
      Full Execution Count=538
      CPU time=120130
      Elapsed Time=120130
      Avg Hard Parse Time=388
      Application Time=0
      Concurrency Time=0
      Cluster/RAC Time=0
      UserI/O Time=0
      Plsql Interpretor Time=0
      JVM Time=0
      Sorts=0
```

可以看出，这个 CURSOR STAT BUCKET 是由 MUTEX 0x660cd02c 来保护的。在 Oracle 10.2 中，只要参数_kks_use_mutex_pin=true，那么 MUTEX 就会被用于 PIN library cache object，在 10.2.0.2 或者更高的版本中，这个参数默认就是 TRUE。在 11*g* 版本中，互斥锁的使用更为广泛，

已知的互斥锁种类包括 CURSOR PARENT、CURSOR PIN、CURSOR STAT、库缓存、散列表等。以往在使用 library cache pin 时，一个闩锁要维护一组对象（比如一组 HASH Bucket），而互斥锁是嵌入到对象内部的，因此一个互斥锁仅仅保护一个特定的对象，这就大大提高了互斥锁并发使用的效率。

互斥锁机制提高了共享池中对象并发访问的效率，特别是提高了游标并发的效率。不过互斥锁也会带来一些新的问题，特别是对于我们 DBA 来说，可能会感到十分的迷茫。如果碰到互斥锁相关的问题，我们该如何处理呢？实际上，Metalink 中的相关文档是十分有用的。

- Troubleshooting “library cache: mutex X” waits. [ID 1357946.1]
- FAQ:“cursor: mutex.. / cursor: pin.. / library cache: mutex ..”Type Wait Events [ID 1356828.1]
- How to Determine the Blocking Session for Event:“cursor: pin S wait on X” [ID 786507.1]
- WAITEVENT: “library cache: mutex X” [ID 727400.1]

根据这几篇文章，老白也对一些主要的等待事件进行了整理，在这里和大家共享，如表 3-3 所示。

表　3-3

等待事件	详细说明
cursor: mutex S cursor: mutex X	说明：以共享模式（S）或者排他模式（X）获取父游标或者访问V$SQLSTAT的时候出现的等待。如果以X模式获取，一般会用来装载子游标或者修改SQL STAT BUCKET。这种等待有三个参数：P1: MUTEX的IDN；P2：MUTEX的值；P3：位置代码（Oracle 开发人员根据这个代码知道是在代码的哪个位置发生的等待，其定义在kgl0.h中） 可能的原因： 频繁的硬解析 有一些游标的版本数量太多 游标存在大量的INVALID和RELOAD Bug导致 查找BLOCKING：在 V$MUTEX_SLEEP_HISTORY中，根据p2参数和REQUESTING_SESSION查找
cursor: pin S cursor: pin X	说明：以共享模式或者排他模式获取游标，由于不兼容访问而出现的等待 可能的原因：同上 查找BLOCKING：同上
library cache: mutex X library cache: mutex X	说明：以共享模式或者排他模式在访问某个库缓存对象时，由于不兼容访问出现的等待 可能的原因：同上 分析方法：参考Troubleshooting“library cache: mutex X”waits. [ID 1357946.1]

3.3　共享池的相关参数

共享池相关的参数如表 3-4 所示（和共享池相关的参数不止下面几个，不同的版本会有所不

同，本表仅列出了一些 DBA 常用的参数）。

表　3-4

参数名称	参数简介	默认值
SHARED_POOL_SIZE	共享池的大小。在10*g*版本之前，该参数控制共享池的大小；在8*i*或者更早的版本中，该参数一旦设置，实例重启前无法修改。从9*i*版本开始可以动态修改（在SGA_MAX_SIZE支持的情况下），从10*g*版本开始这个参数是可以自动管理的，一般情况下可以设置为0，让Oracle 自己管理。不过有经验的DBA一般会给这个参数设置一个值，即共享池的最小值，自动共享内存管理和自动内存管理都无法将共享池缩小到小于这个值的容量	10*g*、11*g*版本默认值为0（使用自动共享内存管理）
SHARED_POOL_RESERVED_SIZE	保留池的大小，默认情况下被设置为共享池的5%。保留池是分配空间给大对象使用的，如果在共享池中无法找到足够大的连续空间，Oracle会从保留池中为其分配对象。Oracle通过一个参数对大对象进行认定（稍后会介绍）	共享池的5%
SHARED_POOL_RESERVED_MIN_ALLOC	设定可以在保留池中分配空间的大对象的最小值。在大多数共享池争用较为严重的系统中，将此参数设置为4000～4200是比较合适的。从8*i*版本开始，该参数变为隐含参数，大家使用时要在前面加上下划线“_”，而且无法通过show parameter显示	4400
CURSOR_SHARING	设置共享游标的策略，默认为EXACT，还可以设置为FORCE或SIMILAR，设置为FORCE或者SIMILAR时，可以强制将未使用绑定变量的SQL合并为一个，从而减少游标的硬解析，提高共享池的性能。当设置为SIMILAR时，在某些条件（比如between、!=、<>）下不会共享游标；当设置为FORCE时，可以强制共享游标	EXACT
OPEN_CURSORS	一个会话可以打开的游标数量的上限。在共享池较为充足的情况下，加大该参数可以改善游标的访问性能	
SESSION_CACHED_CURSORS	一个会话可以缓冲的游标数量的上限，加大此参数，可以减少软解析，进一步提高SQL并发的性能	
_KGHDSIDX_COUNT	共享池的子池数量，Oracle默认根据共享池的大小以及CPU的数量设定，最大为7。如果共享池碎片化程度较严重，不使用默认的配置，减少子池数量，可以对此有所缓解	
SGA_TARGET	严格来讲，SGA_TARGET不能算是共享池的参数，而是SGA的参数，如果设置了SGA_TARGET>0，那么就启用了共享内存自动管理，这样就不需要再设置SHARED_POOL_SIZE等参数了，Oracle会自动进行调整。不过我们还是建议在启用了SGA自动管理（ASMM）的系统中，设置共享池的相关参数（相当于设定其最小值）	

3.4　共享池故障处理

共享池出现故障一般都会导致严重的后果——系统变慢、挂起甚至实例宕掉。由于共享池的

复杂性，其故障分析往往十分复杂，牵涉到的问题很多。因此共享池故障分析也是 DBA 应该掌握的重要技能。

一般来说，共享池故障分为共享池不足、碎片化、闩争用严重等几个方面，本节就从著名的 ORA-4031 错误说起。

3.4.1　著名的 ORA-4031

ORA-4031 是一个十分著名的错误。在共享池中分配内存的时候，共享池无法满足分配需求，从而报错，这就是 ORA-4031。如果这个错误出现在一般性的操作上，那么该操作会失败，而如果出现在核心后台进程的核心操作方面，那么就可能会导致该核心后台进程崩溃，从而使数据库实例宕掉。在 Oracle 10*g* 之前，ORA-4031 是一个十分难缠的问题，而且会带来一系列问题，让很多 DBA 头疼。Oracle 10*g* 共享内存自动管理技术（ASMM）的引入，使 ORA-4031 在 10*g* 系统中的危害大幅减少了。

曾经有朋友问我，在 10*g* 中使用了共享内存自动管理，是不是就永远不会再出现 ORA-4031 了呢？答案当然是否定的，在 10*g* 的数据库中仍然会出现 ORA-4031，代码清单 3-8 是一些来自 10.2.0.4 RAC 的数据。

代码清单 3-8

```
SQL> column indx heading "indx|indx num"
SQL> column kghlurcr heading "RECURRENT|CHUNKS"
SQL> column kghlutrn heading "TRANSIENT|CHUNKS"
SQL> column kghlufsh heading "FLUSHED|CHUNKS"
column kghluops heading "PINS AND|RELEASES"
column kghlunfu heading "ORA-4031|ERRORS"
column kghluNFS heading "LAST ERROR|SIZE"
select
     indx,
     kghlurcr,
     kghlutrn,
     kghlufsh,
     kghluops,
     kghlunfu,
     kghluNFS
  from
    sys.x$kghlu
where
  inst_id = userenv('Instance')
/
      indx  RECURRENT  TRANSIENT    FLUSHED   PINS AND   ORA-4031 LAST ERROR
  indx num     CHUNKS     CHUNKS     CHUNKS   RELEASES     ERRORS       SIZE
---------- ---------- ---------- ---------- ---------- ---------- ----------
         0          0         20          0         20          0          0
         1          0         33          0         33          0          0
         2          0         29          0         29          0          0
         3          0         41          0         41          0          0
         4          0         29          0         29          0          0
         5          0         21          0         21          0          0
```

```
         6          0         25          0         25          0          0
         7      55699      77858  243617195 1849542855      37334       4080
         8      69805      77705  251191900 1951901477      38678       4112
         9      91542     123255  245895667 1903437218      15445       4192
        10      91052     153817  244551473 1575983524      49619       4112
        11      49436      70082  222306066 1187087832      35028       4080
        12      63125     100818  221747017 2048101139      49394       4112
        13      82694     137315  219639529 1724460158      47184       2360
```

从上面的数据可以看出，这是一个 RAC 系统，其中实例 1 的负载很小，SUBPOOL 0 ~ 6 中的 pin 和 release 都很少，不过在 7 ~ 13 这七个子池中都出现过大量的 ORA-4031。为什么在这里有大量的 ORA-4031 出现，但是系统并没有那么多错误呢？这就是 ASMM 的功劳。由于 ASMM 功能的存在，当共享池出现不足的时候，可以通过动态扩展共享池来解决问题，因此即使出现一些 ORA-4031，也不会引起实例宕掉这样的问题。但如果在 9*i* 系统中出现这种情况就麻烦了，当某个核心后台进程遇到了 ORA-4031，必然会导致实例宕掉。

其实在 Oracle 的一些 Metalink 文档中有很全面的关于 ORA-4031 诊断方法的阐述，大家如果有兴趣可以去查阅（读者如果没有 Metalink 账号，可以在 www.oraclefans.cn 论坛发帖，老白会帮助大家上传这些文档）。

- Master Note for Diagnosing ORA-4031 [ID 1088239.1]
- Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]
- Troubleshooting and Diagnosing ORA-4031 Error [Video] [ID 396940.1]

Oracle 也提供了一个诊断工具，帮助用户分析 ORA-4031 问题，拥有 MOS 账号的用户可以直接上传 ALERT LOG、TRACE 文件、AWR 报告等，只需输入下列网址：

```
https://support.oracle.com/CSP/ui/flash.html#tab=Dashboard%28page=GRHome&id=gkzpuq
90%28domainId=ORA4031%29%29
```

进入后，可以按照导航要求上传文件，回答相关问题，最终获得分析结果，如图 3-5 和图 3-6 所示。

图 3-5

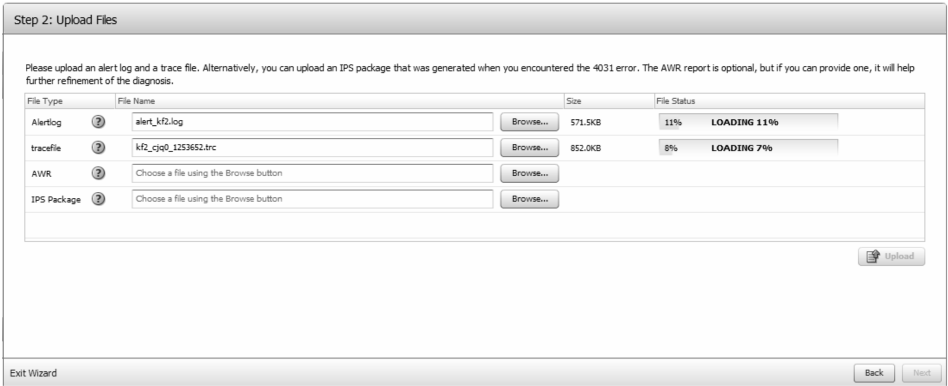


图　3-6

此外，还可以上传 IPS 包（如果你是 11*g* 的用户，ADR 中提供了 IPS 服务——Incident Packaging Services，可以通过 ADRCI 来生成相应的 IPS 包）。上传后，点击 Next 按钮就可以进行分析了，如图 3-7 所示。

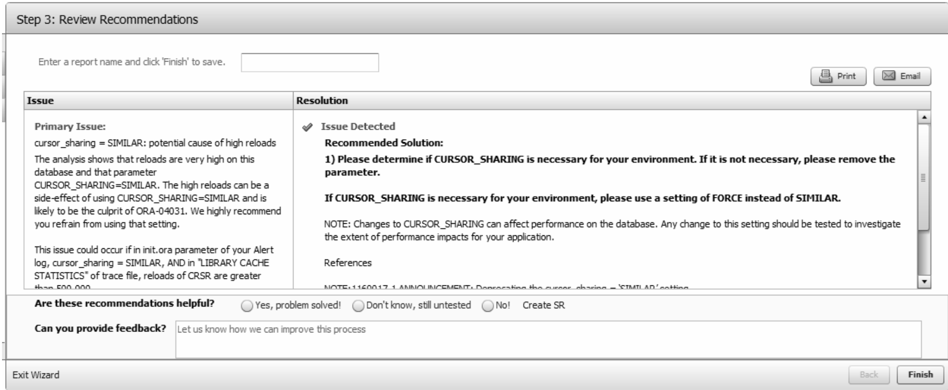


图　3-7

最后查看结果，如图 3-8 所示。

Troubleshooting Report:

Issue	Resolution
Primary Issue: **cursor_sharing = SIMILAR: potential cause of high reloads** The analysis shows that reloads are very high on this database and that parameter CURSOR_SHARING=SIMILAR. The high reloads can be a side-effect of using CURSOR_SHARING=SIMILAR and is likely to be the culprit of ORA-04031. We highly recommend you refrain from using that setting. This issue could occur if in init.ora parameter of your Alert log, cursor_sharing = SIMILAR, AND in "LIBRARY CACHE STATISTICS" of trace file, reloads of CRSR are greater than 500,000 **Evidence Details:** ** In your Alert log, cursor_sharing = SIMILAR, which can cause high library reloads	**Recommended Solution:** **1) Please determine if CURSOR_SHARING is necessary for your environment. If it is not necessary, please remove the parameter.** **If CURSOR_SHARING is necessary for your environment, please use a setting of FORCE instead of SIMILAR.** NOTE: Changes to CURSOR_SHARING can affect performance on the database. Any change to this setting should be tested to investigate the extent of performance impacts for your application. References NOTE:1169017.1 ANNOUNCEMENT: Deprecating the cursor_sharing = 'SIMILAR' setting

图　3-8

我们看到，这个工具给出的最终建议是设置 CURSOR_SHARING=FORCE，这个调整虽然对减少 ORA-4031 有一定帮助，不过针对这个案例，显然是不够全面的。工具可以解决一些问题，但绝对不是问题的全部。虽然 Oracle 提供了 ORA-4031 的诊断工具，但是诊断结果未必准确，因为客户的系统是千差万别的，完全依赖于工具是不可取的。在很多情况下，我们仍然需要手工分析。

一般来说，分析 ORA-4031 需要从 ALERT LOG 和 TRACE 开始，在很多情况下，结合 AWR/STATSPACK 报告是十分必要的。我们还是从上面的那个示例看起，手工分析一下，看看和 ORACLE Metalink 工具分析的结果有何不同。首先查看 ALERT LOG：

```
Wed Apr 30 15:09:37 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_cjq0_1618308.trc:
ORA-00604: error occurred at recursive SQL level 1
ORA-04031: unable to allocate 4032 bytes of shared memory ("shared pool","unknown
object","sga heap(2,0)","kglsim object batch")
Wed Apr 30 15:09:37 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_j000_1581468.trc:
ORA-00604: error occurred at recursive SQL level 1
ORA-04031: unable to allocate 4216 bytes of shared memory ("shared pool","select
u1.user#, u2.user#, u...","sga heap(2,0)","library cache")
Failure to extend rollback segment 11 because of 4031 conditionFULL status of rollback
segment 11 set.
Wed Apr 30 15:09:42 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_j001_1565180.trc:
ORA-00604: error occurred at recursive SQL level 1
ORA-04031: unable to allocate 4216 bytes of shared memory ("shared pool","select
u1.user#, u2.user#, u...","sga heap(2,0)","library cache")
Wed Apr 30 15:09:45 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_smon_1647004.trc:
ORA-00604: error occurred at recursive SQL level 1
ORA-04031: unable to allocate 4216 bytes of shared memory ("shared pool","unknown
object","sga heap(2,0)","library cache")
Wed Apr 30 15:09:46 2008
Errors in file /u01/app/oracle/admin/db10000/udump/kf2_ora_1048804.trc:
ORA-04031: unable to allocate 4032 bytes of shared memory ("shared pool","unknown
object","sga heap(2,0)","kglsim object batch")
Wed Apr 30 15:09:47 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_j001_1565180.trc:
ORA-00604: error occurred at recursive SQL level 2
ORA-04031: unable to allocate 4216 bytes of shared memory ("shared pool","JOB$","sga
heap(2,0)","library cache")
Wed Apr 30 15:09:52 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_j000_1581468.trc:
ORA-00604: error occurred at recursive SQL level 1
ORA-04031: unable to allocate 4216 bytes of shared memory ("shared pool","select
u1.user#, u2.user#, u...","sga heap(2,0)","library cache")
Wed Apr 30 15:09:52 2008
Errors in file /u01/app/oracle/admin/db10000/bdump/kf2_smon_1647004.trc:
ORA-00604: error occurred at recursive SQL level 1
ORA-04031: unable to allocate 4216 bytes of shared memory ("shared pool","unknown
object","sga heap(2,0)","library cache")
```

```
Wed Apr 30 15:09:55 2008
Shutting down Instance (abort)
License high water mark = 151
Instance terminated by USER, pid = 1048804
```

这个系统出现了大量的 ORA-4031，主要是无法分配一些大于 4KB 的空间，最终 DBA 通过 shutdown abort 强制关闭后重启了系统。从 ALERT LOG 中我们并没有发现更多有价值的内容，接下来查看 TRACE 文件：

```
*** 2008-04-30 13:24:47.918
*** SESSION ID:(19.1) 2008-04-30 13:24:47.904
=================================
Begin 4031 Diagnostic Information
=================================
The following information assists Oracle in diagnosing
causes of ORA-4031 errors.  This trace may be disabled
by setting the init.ora parameter _4031_dump_bitvec = 0
======================================
Allocation Request Summary Information
======================================
Current information setting:  00654fff
  Dump Interval=300 seconds  SGA Heap Dump Interval=3600 seconds
  Last Dump Time=04/30/2008 13:24:46
Allocation request for: kglsim object batch
Heap: 70000000004b848, size: 4032
```

从最后两行可以看出，在分配 kglsim 对象的时候，无法分配 4032B 的连续空间，因此报错。接下来查看共享池的整体情况：

```
===============================
Memory Utilization of Subpool 1
===============================
     Allocation Name                Size
___________________________       ________
"free memory"                     29313288
"miscellaneous"                   12936120
"transaction"                       800448
"UNDO INFO SEGMENTED ARRAY"         325056
"errors"                             23080
"temporary tabl"                      3136
"SEQ S.O."                          264800
"partitioning d"                     92400
"db_handles"                       1740000
"replication session stats"         503120
"ges regular msg buffers"          1576248
"table definiti"                       776
"PL/SQL MPCODE"                     419304
"gcs resource hash table"          2097152
"PL/SQL DIANA"                      608352
"trigger inform"                         0
"ges enqueues"                     6563240
"PL/SQL PPCODE"                          0
"ges resource hash table"          4325376
```

```
"trigger defini"                      0
"gcs resources"               45304256
"sim memory hea"               2558400
"dictionary cache"             1065728
"db_block_hash_buckets"       19589168
"ges resources"              441539912
"KQR M PO"                       45568
"Checkpoint queue"             5245440
"library cache"                6486264
"type object de"                     0
"sql area"                     1229936
"sessions"                     1119456
"gcs shadows"                 28805632
"event statistics per sess"    4696416
"trigger source"                     0
"VIRTUAL CIRCUITS"              926800
"KGLS heap"                     517760
"parameters"                     34048
"fixed allocation callback"        312
===============================
Memory Utilization of Subpool 2
===============================
     Allocation Name          Size
___________________________    __________
"free memory"               7120752
"miscellaneous"            31854176
"PL/SQL DIANA"               327896
"gcs resources"            45304256
"Checkpoint queue"          5245440
"sim memory hea"            2558400
"PL/SQL PPCODE"                   0
"KQR S SO"                     7952
"PL/SQL MPCODE"              103384
"ges enqueues"              6085384
"ges resources"           441419824
"FileOpenBlock"            15431456
"parameters"                      0
"joxs heap init"               1424
"ges big msg buffers"       6770088
"db_block_hash_buckets"     2490368
"enqueue"                   3177816
"gcs shadows"              28802536
"temporary tabl"               6480
"PLS non-lib hp"               2744
"KGK heap"                    32816
"KCL name table"            7345992
"dictionary cache"          2103808
"partitioning d"                  0
"trigger inform"                  0
"KGLS heap"                   29080
"trigger source"                  0
"library cache"             7696608
"processes"                 1968000
"sql area"                    28072
```

```
"PL/SQL SOURCE"                        0
"KQR M PO"                        156864
"errors"                               0
"trigger defini"                       0
"event statistics per sess"      4685072
"fixed allocation callback"          304
"type object de"                       0
"table definiti"                       0
===============================
Memory Utilization of Subpool 3
===============================
     Allocation Name             Size
___________________________    ________
"free memory"                   17748320
"miscellaneous"                 17136616
"ges enqueues"                   6542112
"gcs shadows"                   28803528
"table definiti"                       0
"trigger source"                       0
"parameters"                           0
"sim memory hea"                 2568592
"KQR M PO"                        184376
"partitioning d"                       0
"errors"                               0
"PL/SQL MPCODE"                   116192
"ges reserved msg buffers "      2096008
"gcs resources"                 45304480
"KGLS heap"                         7840
"temporary tabl"                    4240
"ktlbk state objects"             975520
"PL/SQL PPCODE"                        0
"MTTR advisory"                    38448
"KSXR pending messages que"       853952
"KQR L PO"                        151552
"db_block_hash_buckets"         19589184
"dictionary cache"               1071104
"ges resources"                441479424
"enqueue resources"              1084528
"PL/SQL SOURCE"                        0
"PX subheap"                       30296
"library cache"                  6153192
"KGK heap"                           552
"sql area"                         76136
"sessions"                       1119456
"trigger inform"                    1624
"event statistics per sess"      4696416
"PL/SQL DIANA"                         0
"trigger defini"                       0
"Checkpoint queue"               5245440
"fixed allocation callback"          328
"VIRTUAL CIRCUITS"                900320
===============================
Memory Utilization of Subpool 4
===============================
```

```
      Allocation Name              Size
____________________________    ________
"free memory"                    6669520
"miscellaneous"                 20107808
"KQR M PO"                          8704
"sim memory hea"                 2560968
"errors"                               0
"KQR L PO"                        129024
"KCL bast context freelist"       336000
"session param values"           3621888
"gcs resources"                 60737168
"transaction"                     667040
"table definiti"                       0
"PL/SQL DIANA"                     40200
"trigger defini"                       0
"KGLS heap"                        26560
"partitioning d"                       0
"MTTR advisory"                   606000
"parameters"                        1064
"trigger inform"                       0
"ges resources"                436017912
"channel handle"                  391504
"trigger source"                       0
"ges enqueues"                   6048136
"dictionary cache"                 33792
"PL/SQL PPCODE"                        0
"gcs shadows"                   44234376
"KCL lock context"                364320
"temporary tabl"                    2504
"library cache"                  6557816
"Temporary Tables State Ob"       389736
"sql area"                         42920
"sessions"                       1119456
"PL/SQL MPCODE"                   111424
"event statistics per sess"      4696416
"FileIdentificatonBlock"         1153752
"DML lock"                       1529560
"1M buffer"                       528384
"Checkpoint queue"               5245440
"fixed allocation callback"          384
```

可以看出，这个系统的共享池分为 4 个子池，每个子池的空闲内存容量都比较大，最大的有 200 多兆字节，最小的有 60 多兆字节。这说明共享池的空闲空间充足，只是碎片化比较严重。从上面的数据我们看到库缓存和 sql area 所占的空间其实并不大，这一点可以说明，MOS 分析给出的修改 CURSOR_SHARING 参数的方案并不是真正的核心。虽然调整 CURSOR_SHARING 可以改善共享池，但是其根本原因并不在这里。从 ges resources 项的数据中可以看出一些比较异常的情况。在每个子池中，ges resources 都占用了 440MB 左右，这是相当大的。这是一套 RAC 系统，ges resources 用于 GLOBAL ENQUEUE SERVICE 相关的数据结构，一般情况下在系统启动时根据 lm_locks 和 lm_ress 参数的值以及 DB CACHE 的大小确定初始化分配的容量，通常为几十兆字节。当这种数据结构不够用的时候，会自动进行动态扩展。在 Oracle 9*i* 中，这项资源的扩展方式

是每次扩展一个。从 Oracle 10.2 开始，为了防止这种扩展方式导致共享池碎片化，算法改为每次扩展一组，不再逐个扩展。ges resource 资源是 PERMANENT 的对象，扩展后基本不会释放。因此如果 ges resources 扩展得十分频繁，就会在共享池中形成很多不可释放的点，从而将共享池截断为多个小块，造成共享池碎片化。在刚刚重启的系统中，我们检查了 ges resources 的大小。在 4 个子池中，ges resources 资源总计为 100 多兆字节，这说明 ges resources 的扩展十分频繁，这样就可能导致共享池由于 ges resources 扩展而形成大量的碎片。因此初步定位为 ges resources 导致了问题。想要在 9*i* 中彻底解决这个问题是比较困难的，而在 10*g* 中，由于 ges resources 扩展算法的优化，这种问题就不容易出现了。不过在 9*i* 中，可以通过增加共享池的容量，增加 lm_locks 和 lm_ress 参数来缓解此问题。调整参数后，系统中的问题确实得到了解决。

在分析和解决 ORA-4031 的问题时，最主要的是分析问题发生的原因，到底是由于共享池容量不足还是由于共享池碎片化。如果共享池的空闲内存数量很小，那么很可能是共享池容量不足导致，而如果空闲内存比较大，但还是经常会出现 ORA-4031，那么共享池碎片化导致该问题的可能性就比较大了。这种情况下，查看每个对象的大小，确认是否存在异常是十分关键的。如果能够找到一些特别之处，我们就可以根据这个线索往下追溯，从而定位问题的关键。比如，如果我们发现 sql area 和库缓存所占的空间较大，那么就可以分析如何减少它们所占空间的大小；如果我们发现存在大量的高版本的 SQL，那么将 CURSOR_SHARING 从 SIMILAR 修改为 FORCE 可能是比较好的方案。

如果ORA-4031主要是因为共享池设置偏低导致的,那么扩大共享池就是十分好的解决方案。可能很多 DBA 认为使用绑定变量是最佳的解决方案，不过最好的方案往往是可望而不可及的。修改应用是一个十分敏感的话题，也意味着大量的成本投入，能够通过扩大内存解决的问题，有可能修改应用是很难被接受的。

如果是由于共享池碎片导致的 ORA-4031 问题，那么减少子池的数量，以及调整使用保留池的最小分配内存参数（比如从默认的 4400 调整为 4000）是十分有效的手段。

随着共享内存自动管理技术和内存自动管理技术的引入以及内存价格的大幅度下降，ORA-4031 这个问题好像不那么棘手了。由于共享池可以动态地自动扩展，因此即使出现了 ORA-4031，也不会引起十分严重的问题。不过共享内存自动管理也有其局限性，对于负载很高，或者负载波动十分频繁的系统，如果 SGA_TARGET 设置得偏低，那么就很容易出现 SGA RESIZE。在业务十分繁忙的时候出现较为频繁的 SGA RESIZE 会严重影响 SGA 的性能，导致游标方面的争用，严重时会导致系统挂起。因此如果想要使用共享内存自动管理，那么首先要确保 SGA_TARGET 的设置是充足的，甚至略高于系统最高负载。另外要设置共享池的最小值（设置 SHARED_POOL_SIZE 参数），这样才能确保 SGA 的效率不会因为 SGA RESIZE 而大幅度下降。

虽然在共享内存自动管理的情况下，ORA-4031 的出现不再那么危险，不过经常监控是否存在 ORA-4031，定期在系统较空闲的时候刷新一下共享池，对于共享池的高效运作还是很有帮助的。

3.4.2　其他共享池常见故障

共享池响应速度下降是很常见的共享池故障。从 AWR 报告中，我们经常会看到类似下面的问题。

```
Event                        Waits  Time(s) Avg Wait(ms)  % Total Call Time  Wait Class
----------------------- ------- ------- ------------ ----------------- ----------
CPU time                          10,018                        78.3
latch: library cache      3,200      880        275              6.9      Concurrency
latch: shared pool        3,510      849        242              6.6      Concurrency
latch free                1,186      305        257              2.4            Other
db file sequential read  62,915      282          4              2.2         User I/O
```

在上面的示例中，共享池相关闩锁的总等待时间虽然不到 20%，但是平均等待时间却超过了 200 毫秒，而这些闩锁的正常等待时间应为几毫秒，因此系统的性能会受到很大的影响。在这种情况下，需要分析到底是什么原因导致了这个问题。以老白的经验来看，导致类似问题的可能性最大的因素包括：共享池过小，共享池 RESIZE 操作，有人在编译重要的存储过程或者视图，碰到了 Bug。

共享内存自动管理或者内存自动管理中共享池 RESIZE 操作频繁出现导致类似问题的可能性很大，可以通过查询 V$SGA_RESIZE_OPS 视图来进行诊断，下面是一个存在共享池抖动的典型案例。

```
COMPONENT            OPER_TY  OPER_MODE    START_TIME         END_TIME        TARGET
------------------- -------- --------- ------------------ ------------------ -----
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:19 2009-08-25 11:01:19 20528
shared pool          GROW    IMMEDIATE 2009-08-25 11:01:19 2009-08-25 11:01:19  1824
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:39 2009-08-25 11:01:39 20512
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:39 2009-08-25 11:01:39    48
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:41 2009-08-25 11:01:41 20496
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:41 2009-08-25 11:01:41    64
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:47 2009-08-25 11:01:47 20480
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:47 2009-08-25 11:01:47    80
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:48 2009-08-25 11:01:48 20464
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:48 2009-08-25 11:01:48    96
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:50 2009-08-25 11:01:50 20448
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:50 2009-08-25 11:01:50   112
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:53 2009-08-25 11:01:53 20432
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:53 2009-08-25 11:01:53   128
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:57 2009-08-25 11:01:57 20416
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:57 2009-08-25 11:01:57   144
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:58 2009-08-25 11:01:58 20400
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:58 2009-08-25 11:01:58   160
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:01:59 2009-08-25 11:01:59 20384
streams pool         GROW    IMMEDIATE 2009-08-25 11:01:59 2009-08-25 11:01:59   176
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:02:00 2009-08-25 11:02:00 20368
streams pool         GROW    IMMEDIATE 2009-08-25 11:02:00 2009-08-25 11:02:00   192
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:02:01 2009-08-25 11:02:01 20352
streams pool         GROW    IMMEDIATE 2009-08-25 11:02:01 2009-08-25 11:02:01   208
DEFAULT buffer cache SHRINK  IMMEDIATE 2009-08-25 11:02:09 2009-08-25 11:02:09 20336
streams pool         GROW    IMMEDIATE 2009-08-25 11:02:09 2009-08-25 11:02:09   224
```

```
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:02:09 2009-08-25 11:02:09 20320
streams pool          GROW    IMMEDIATE 2009-08-25 11:02:09 2009-08-25 11:02:09   240
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:02:10 2009-08-25 11:02:10 20304
streams pool          GROW    IMMEDIATE 2009-08-25 11:02:10 2009-08-25 11:02:10   256
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:02:11 2009-08-25 11:02:11 20288
streams pool          GROW    IMMEDIATE 2009-08-25 11:02:11 2009-08-25 11:02:11   272
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:02:12 2009-08-25 11:02:12 20272
streams pool          GROW    IMMEDIATE 2009-08-25 11:02:12 2009-08-25 11:02:12   288
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:02:13 2009-08-25 11:02:13 20256
streams pool          GROW    IMMEDIATE 2009-08-25 11:02:13 2009-08-25 11:02:13   304
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:02:20 2009-08-25 11:02:20 20240
streams pool          GROW    IMMEDIATE 2009-08-25 11:02:20 2009-08-25 11:02:20   320
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:03:17 2009-08-25 11:03:17 20224
streams pool          GROW    IMMEDIATE 2009-08-25 11:03:17 2009-08-25 11:03:17   336
streams pool          GROW    IMMEDIATE 2009-08-25 11:03:20 2009-08-25 11:03:20   352
DEFAULT buffer cache  SHRINK  IMMEDIATE 2009-08-25 11:03:20 2009-08-25 11:03:20 20208
streams pool          GROW    IMMEDIATE 2009-08-25 11:03:21 2009-08-25 11:03:21   368
```

为了避免占用过多篇幅，老白省去了大部分的数据，原始查询结果有800行。不过从这些数据已经可以看出，在3分钟多的时间内，SGA的各个池在不停变化，这么频繁的变化肯定会导致共享池出现严重的性能问题。

除了共享池抖动外，还有一个经常出现的问题——共享池突发挂起现象。这种现象包括几种情形，最常见的一种是其他操作都很正常，只是对某张表的操作出现问题，哪怕只是简单的SELECT操作也会导致挂起。碰到这种情况如何分析呢？最快捷和简单的方法是通过v$session_wait检查会话在等待什么事件。比较合理的做法是，先进行HANGANALYZE分析，查看到底系统是否存在挂起现象。这里介绍一个相关的案例。

一个客户的系统突然出现了大量会话挂起的现象，系统中的会话数突然猛增。DBA经过检查发现挂起的会话都在等待行锁。直接对这个行锁涉及的表执行SELECT操作也会导致该表挂起。接到电话后，老白马上让DBA做了一个3级的HANGANALYZE分析，通过HANGANALYZE报告我们发现：

```
HANG ANALYSIS:
==============
Found 36 objects waiting for <sid/sess_srno/proc_ptr/ospid/wait_event>
    <153/8226/0xbd414b0/25567/enqueue>

Cycle 1 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <61/60457/0xbcd9640/29261/library cache lock>
 -- <153/8226/0xbd414b0/25567/enqueue>
Open chains found:
Chain 1 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <375/41785/0xbd13c10/29658/No Wait>
Chain 2 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <491/62836/0xbd6cbd0/7054/No Wait>
Chain 3 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <676/34256/0xbd71300/13956/No Wait>
Chain 4 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <743/13745/0xbd55720/16135/SQL*Net break/reset to client>
Other chains found:
```

```
Chain 5 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <47/12295/0xbd0be70/29623/library cache pin>
Chain 6 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <68/4549/0xbd1b150/29707/library cache lock>
 -- <620/14459/0xbd844b0/6103/library cache lock>
 -- <299/29643/0xbd7adf0/7060/library cache lock>
Chain 7 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <89/35994/0xbd44b20/6461/library cache lock>
 -- <620/14459/0xbd844b0/6103/library cache lock>
 -- <299/29643/0xbd7adf0/7060/library cache lock>
Chain 8 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <116/2726/0xbd1b9b0/29711/library cache lock>
 -- <620/14459/0xbd844b0/6103/library cache lock>
 -- <299/29643/0xbd7adf0/7060/library cache lock>
Chain 9 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <131/62101/0xbd85140/6201/enqueue>
Chain 10 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <135/42351/0xbd19c60/29699/library cache lock>
 -- <620/14459/0xbd844b0/6103/library cache lock>
 -- <299/29643/0xbd7adf0/7060/library cache lock>
```

为了节省篇幅，此处略去了分析结果的后半部分，不过前面部分已经足够让我们完成这次分析了。首先可以看到：

```
Found 36 objects waiting for <sid/sess_srno/proc_ptr/ospid/wait_event>
    <153/8226/0xbd414b0/25567/enqueue>
```

SID=153 的会话在等待 enqueue，而有 36 个其他的会话在等待这个会话持有的资源。

似乎一切都很简单，我们已经发现了问题的“元凶”，在这种情况下，是不是杀掉 153 号会话就能够解决问题了呢？答案当然是否定的，一名优秀的 DBA 绝对不会这样做，我们必须继续往下分析 HANGANALYZE 报告。

```
Cycle 1 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <61/60457/0xbcd9640/29261/library cache lock>
 -- <153/8226/0xbd414b0/25567/enqueue>
```

无论如何，Cycle 是必须分析的，因为 Cycle 往往是能够找到问题根源的地方。Cycle 1 的“受害者”是 153 号会话，也就是阻塞了 36 个会话的“元凶”，而 153 号会话在等待 enqueue，这个 euqneue 却被 61 号会话所持有，而 61 号会话正在等待 library cache lock。我们在 HANGANALYZE 报告中检查了一遍，并没有会话阻塞 61 号会话。在这种情况下，如果是很紧急的生产系统故障，那么杀掉 61 号会话，看系统能否恢复是比较简单的处理方法。而如果我们要去追寻问题的根源所在，就需要继续分析，这时系统状态转储是十分有用的。不过在一个特别重要的生产系统中，可能不允许我们进行系统状态转储，这样就无法进一步分析问题的原因。比如上例中，我们征询了客户的意见，然后做了一个 LEVEL 266 的系统状态转储，发现 SESSION 61 在等待一个 library cache lock，而这个 library cache lock 正被自己持有，这是典型的自死锁现象。于是我们立即杀掉 SESSION 61（从 HANGANALYZE 报告中可以看到这个会话的 OSPID 是 29261)，解决了这个问题。

共享池的问题极为复杂，不是短短的几节内容就能够涵盖的。除了共享池内存不足和碎片化这两个最为典型的问题，大多数情况下共享池出现问题是由于游标共享或者并发访问某个共享池对象引起的。下面的几篇 Metalink 文档对于分析共享池问题有很好的帮助，建议大家仔细阅读，并随身携带。

- ❑ Troubleshooting Library Cache: Lock, Pin and Load Lock [ID 444560.1]
- ❑ Troubleshooting: “WAITED TOO LONG FOR A ROW CACHE ENQUEUE LOCK! ”[ID 278316.1]
- ❑ Troubleshooting: High Version Count Issues [ID 296377.1]
- ❑ How to Determine the Blocking Session for Event: “cursor: pin S wait on X” [ID 786507.1]
- ❑ Troubleshooting “library cache: mutex X” waits. [ID 1357946.1]
- ❑ How To Troubleshoot ORA-4031’s and Shared Pool Issues With Procwatcher (Doc ID 1352623.1) [ID 1355030.1]
- ❑ Troubleshooting: Tuning the Shared Pool and Tuning Library Cache Latch Contention [ID 62143.1]
- ❑ Troubleshooting: Waits for Mutex Type Events [ID 1377998.1]
- ❑ Troubleshooting: Tuning the Shared Pool and Tuning Library Cache Latch Contention [ID 62143.1]

3.5　共享池优化的主要思路

共享池的优化要从几个方面去考虑：首先是共享池本身的配置，其次是游标的共享，然后是缓解共享池的碎片，最后是分析 Bug 和补丁。

共享池配置主要是看共享池的容量是否足够。10 多年前，DBA 管理共享池是十分痛苦的，由于物理内存总是无法满足应用系统的需要，因此总是无法分配给共享池足够的空间，我们必须想尽一切办法来减少应用对共享池的使用，从而使更多的并发访问能够尽可能地对共享池产生最小的影响。而共享池管理总是希望某个字典缓存或者库缓存能够在共享池中存储更多的时间。在目前的硬件条件下，物理内存几乎是无极限的（由于内存的廉价和服务器可配置内存总量的大幅提升），因此我们总是可以给共享池分配足够的空间，从而保证共享池的高效运作。在现阶段来看，保持共享池配置有足够的空间是共享池优化工作中十分重要的一点。

在共享池的配置中，子池的数量也是一个十分重要的因素。在 11*g* 版本之前，共享池子池的最小值偏小，而由于目前多核 CPU 的广泛使用，使得 Oracle 识别的逻辑 CPU 总数很多，基于每 4 个 CPU 分配一个子池的原则，很容易自动分配较多的子池。如果共享池偏小，而且碎片化比较严重，那么减少子池的数量，使每个子池不小于 256MB 甚至 512MB 是十分必要的。

在游标共享方面，尽可能使用绑定变量是非常重要的。共享游标可以减少大量的共享池空间，减少共享池中游标的总数量，从而减少共享池的争用。通过使用绑定变量和良好的 SQL 书写风格可以增加游标的共享。如果应用程序未使用绑定变量，我们也可以通过将 CURSOR_SHARING

参数设置为FORCE或者SIMILAR来加强游标的共享。很多专家都建议使用CURSOR_SHARING=FORCE，而不建议使用SIMILAR。这是因为SIMILAR可能出现另外的问题，使某个游标出现大量不能共享的子游标。实际上FORCE也有其自身的问题，如果某条SQL在参数不同时需要使用不同的执行计划，那么使用了FORCE之后，就存在问题了（在11*g*版本中由于ACS的出现，这个问题已经得到了很明显的改善），某些SQL执行计划的错误相比共享池中的游标不能共享来说，可能具有更大的危害，在这方面我们需要做好权衡。

基于上述原因，老白建议在11*g*之前的版本中，还是通过适当地使用绑定变量来解决游标的共享问题。在某些需要根据柱状图的不同而采用不同执行计划的情况下，尽可能不使用绑定变量。而对于11*g*版本的系统，尽可能使用绑定变量吧（在使用时要注意，如果存在较多SQL版本，可能是碰到了ACS的Bug）。

游标共享可以减少硬解析的数量，从而缓解共享池相关争用。在有些情况下，调用的次数很多，从而导致接卸的数量很大，达到每秒几千甚至上万。在这种情况下，减少软解析也是十分关键的，通过加大OPEN_CURSORS和SESSION_CACHED_CURSORS参数，尽可能保持热点游标处于开放和缓存状态，减少软解析的数量，可以有效地缓解共享池的性能问题，加大系统并发处理能力。

如果共享池碎片化十分严重，就需要了解碎片产生的原因，从根本上解决问题。如果暂时无法找到问题的根源，也可以通过定期在业务较小的时候手工刷新共享池来保持共享池的效率，减少由于共享池碎片化而导致的性能问题。为了缓解碎片化的问题，将一些重要的PL/SQL对象和SQL保留在共享池中也是十分有效的方法。特别是一些较大的PL/SQL对象，每次数据库重启后，通过DBMS_SHARED_POOL.KEEP存储过程将其保留在共享池中，可以减少这些大型对象重新加载的次数，从而缓解其加载过程对共享池的影响。

为了减少共享池产生性能问题，在业务高峰期尽可能不要修改表结构也是十分重要的，在业务高峰期进行表和索引的修改、授权等操作，可能导致共享内存中大量的游标失效，从而产生大量的硬解析，严重时甚至会导致挂起。

从10*g*版本开始，由于共享内存自动管理的引入，共享池的优化工作也简化了许多，只要设置足够大的SGA_TARGET参数，共享内存就可以被Oracle自动管理了。不过使用共享内存自动管理（或者11*g*版本的内存自动管理）也不能一劳永逸地解决共享池问题。在某些情况下，如果SGA_TARGET不足，共享内存可能出现抖动，从而造成严重的共享池问题。因此在使用共享内存自动管理时不能太过简单，除了设置SGA_TARGET外，还需要设置共享池的SHARED_POOL_SIZE参数。通过该参数，设置共享池的最小值，将这个值设置得足够大，可以确保共享池不会由于SGA抖动而出现严重的性能问题。

另外很多共享池的性能问题是由于Bug引起的，在安装数据库的时候，尽可能用最新的补丁集，减少由于Bug导致问题的可能性。不过安装最新的补丁包并不等于高枕无忧，定期分析ALERT LOG，分析AWR报告，及时发现新的问题是十分必要的。

第 4 章 理解控制文件

控制文件是 Oracle 数据库中一个十分重要的文件，虽然这个文件不大，而且访问它的 I/O 量也不是很大，但是这个文件中却包含了整个数据库的结构。控制文件中的信息包含数据库、数据文件、CHECKPOINT、REDO LOG 及归档日志信息和 RMAN 备份的信息。这些信息对于数据库是十分重要的，对于数据库实例恢复、介质恢复等都具有关键的作用。

如果控制文件出现故障，可能导致数据库无法正常打开，Oracle 数据库通过多个控制文件副本的方式来实现控制文件的高可用性。Oracle 对这些控制文件副本采用并行写的方式，所有的控制文件操作都在一个逻辑的控制文件事务中完成对所有控制文件副本的更新。只需将这些控制文件副本存储在不同的文件系统或者磁盘上，即使某个文件系统或者磁盘出现故障，也不会损坏所有的控制文件，只要有一个控制文件的副本是可用的，我们就可以用这个副本来打开数据库。

由于控制文件并行写是通过一个逻辑事务完成的，在一个逻辑事务中有多个物理 I/O，因此当系统宕掉的时候，可能会出现这种情况：某个控制文件操作在一个控制文件中已经完成，而另一个控制文件和这个控制文件不一致。这种情况下，会出现控制文件损坏的现象。我们只要用完好的控制文件覆盖损坏的控制文件，就可以解决问题了。

4.1 控制文件的内部结构

控制文件相对较为简单，所以大家对控制文件内部结构的研究也较为透彻，在网络上有大量关于控制文件结构的资料，一般来说，了解了这些资料，就足以支撑 DBA 的日常维护工作了。

本节重点关注控制文件事务方面的技术细节，对于那些 METALINK 和网络上介绍较多的控制文件内部数据项只做简单的介绍。

4.1.1 控制文件和控制文件事务

控制文件分为文件头和内容两部分。文件头包含控制文件块大小、控制文件包含的数据块数量这些信息。在加载数据库的时候，启动进程会读取控制文件的头块，并进行校验。如果校验失败，数据库会报错。在 Oracle 10*g* 之前，控制文件的块大小一般为 4096B，从 10*g* 版本开始，控

制文件的块大小被扩大为 16KB。

控制文件中保存的最为重要的数据是整个数据库的结构，包括所有的数据文件、在线日志文件等，因此控制文件是数据库能够正常打开的关键。如果控制文件损坏，那么 RDBMS 就无法知道数据库所属文件的位置以及数据文件的状态，数据库也就无法正常工作。

控制文件的更新操作实际上有一些类似于事务的特点，有些操作必须一起完成或者一起失败。而针对控制文件的操作是不适合使用类似普通事务的方式的，因此针对控制文件事务也需要一套机制来确保其完整性。而且在数据库实例突然宕机或者操作控制文件的进程出现故障的时候，需要有一套机制来确保被中断的操作不会损坏控制文件里的数据项。控制文件的操作采用了较为简单的方法，通过一个 CF 锁来进行排他操作。如果要进行控制文件事务，那么就需要持有 CF 锁的排他模式，从而避免并发的同类操作。对于读操作，只需要共享的 CF 锁，因此两个并行的读操作是可以同时进行的。

控制文件的前部包含一个数据库信息的记录，这个记录大概在 210B 左右（不同的数据库版本可能略有不同），不过这些信息还是占用了一个独立的控制文件块。控制文件这样的处理方式降低了控制文件事务的操作复杂度，因为一个独立的控制文件库可以通过一个独立的原子操作来完成修改，这种设计也就简化了错误恢复。Oracle 为了实现控制文件事务的故障回滚，设计了一种很巧妙的算法。所有的控制文件信息都是双份存储的，任何一个逻辑记录都有两个物理的副本。其中的一个副本是当前副本，也就是当前正在使用的副本；而另外一个副本是旧副本或者正在进行修改还没提交的副本。在控制文件的数据库信息记录里保存了一个控制位图信息，用以记录哪个副本是当前正在起效的副本。如果要读取控制文件的信息，会话必须首先读取版本控制位图确定哪个物理块是当前的，然后再去读取这个数据。

当会话要去修改某个数据的时候，首先需要以排他模式获取 CF 锁，这样就避免了其他会话同时修改控制文件。获得 CF 锁后，会话首先会修改非当前的数据块的信息，修改完成后，通过一个原子操作将控制位图信息中的当前块信息更新，这样修改就完成了。

如果一个修改操作需要修改某个控制文件块中的多个记录，那么 Oracle 会首先将这些修改存储在一个缓冲区中，等数据组织完成后，一起写入控制文件，从而提高写的效率。根据上面的算法，每个控制文件事务至少需要 4 个串行的 I/O 操作，如果存在多个控制文件副本，那么这些 I/O 操作有一半会以异步 I/O 的方式进行，如果系统不支持异步 I/O，那么 I/O 操作的数量会成倍增加。而如果一个控制文件事务要修改多个控制文件块中的数据，I/O 的数量就更大了。所以说控制文件事务是一种 I/O 敏感的操作，如果控制文件存放在性能不好的存储上，那么控制文件方面的锁等待将会很严重。从这一点可以看出，我们以前的一些观点是错误的。以往我们认为控制文件的读写量很小，因此可以将它们存放在 I/O 性能相对较差的存储设备上，在了解了控制文件事务的特点后，就不会再这样处理了。如果存储控制文件的设备 I/O 性能不佳，那么 CF 锁等待将会对系统的性能造成影响。

视图 V$CONTROLFILE_RECORD_SECTION 包含了所有的控制文件结构。这个视图是对内部数据结构 X$KCCRS 中存储信息的重新整理。相关示例如下：

```
NAME                RSLBN    RSRSZ      RSNUM     RSNUS      RSIOL     RSILW    RSRLW
--------------- -------- ---------- --------- ---------- --------- -------- ------
DATABASE              1        192         1          1          0        0        0
CKPT PROGRESS         2       4084         4          0          0        0        0
REDO THREAD           4        104         1          1          0        0        0
REDO LOG              5         72         5          3          0        0        3
DATAFILE              6        180       100         11          0        0      145
FILENAME              9        524       116         15          0        0        0
TABLESPACE           17         68       100         12          0        0       12
TEMPORARY FILENAME 18           56       100          1          0        0        1
RMAN CONFIGURATION 19         1108        50          0          0        0        0
LOG HISTORY          26         36       226         28          1       28       28
OFFLINE RANGE        27         56       145          0          0        0        0
ARCHIVED LOG         28        584        13          4          1        4        4
BACKUP SET           29         40       204          0          0        0        0
BACKUP PIECE         30        736       210          0          0        0        0
BACKUP DATAFILE      49        116       211          0          0        0        0
BACKUP REDOLOG       52         76       107          0          0        0        0
DATAFILE COPY        53        660       210          0          0        0        0
BACKUP CORRUPTION    70         44       185          0          0        0        0
COPY CORRUPTION      71         40       204          0          0        0        0
DELETED OBJECT       72         20       408          0          0        0        0
PROXY COPY           73        852       210          0          0        0        0
RESERVED4            96         36       226          0          0        0        0
                     97        276         1          1          0        0        0
                     98         56       145          1          1        1        1
```

通过上面的结果，读者可以了解控制文件中的主要内容、控制文件中包含数据库的信息、CKPT 的信息、REDO LOG 的信息、数据文件的信息、表空间的信息以及日志切换历史信息。除此之外，控制文件中还包含 RMAN 备份的 CATALOG 信息。通过对上述信息的分析，我们可以了解 RMAN 在不使用恢复目录数据库的时候，是如何把信息存储在控制文件中的。

CHECKPOINT 的进度信息也会被保存在控制文件中，而且 CHECKPOINT 是发生频率很高的操作。如果更新 CHECKPOINT 进度信息也需要使用控制文件事务，那么 CKPT 进程将会成为 CF 锁的长期持有者，这样就增加了系统由于 CF 锁冲突而挂起的可能性，因此 CHECKPOINT 信息的修改不使用控制文件事务机制。CHECKPOINT 进度记录占用了一个控制文件块的一半容量，因此 Oracle 为每个数据库实例分配一个独立的物理块来记录 CHECKPOINT 进度记录。和普通的控制文件块不同，CHECKPOINT 进度记录使用单一的物理块来记录数据。这样的机制使 CHECKPOINT 进度的写入可以作为一个原子操作，进行该操作时，不会影响其他数据。

4.1.2　控制文件自动扩展

在控制文件中，如果已经存在的数据要重新写入新的内容，那么 Oracle 数据库会重用这个记录。大多数控制文件记录都可以循环使用，比如 RMAN CATALOG 的信息和 LOG 历史信息。Oracle 通过一个初始化参数 control_file_record_keep_time 来设定控制文件中的数据至少保存多少天才能够被重用，它的默认值是 7。这个参数可以控制的控制文件记录包括一系列周期性的数据：LOG HISTORY、OFFLINE RANGE、ARCHIVED LOG、BACKUP SET、BACKUP PIECE、BACKUP DATAFILE、BACKUP REDOLOG、DATAFILE COPY、BACKUP CORRUPTION、COPY

CORRUPTION、DELETED OBJECT 和 PROXY COPY。

如果所有的空闲记录都已经写满了，但是最旧的记录还没有保存到参数设定的天数，那么Oracle 会动态扩展记录节的大小（如果控制文件不足以提供记录节扩展所需要的空间，控制文件也会动态扩展），每个记录节的最大记录数是 65535，因此控制文件动态扩展时不能超过这个限制。控制文件的大小也有一定的限制，控制文件的大小受到块版本位图大小的限制，由于块版本位图都很大，这个限制一般很难达到。如果控制文件进行了动态扩展，那么在 ALERT LOG 中会出现一些 kccrz 的信息。要避免这种扩展，可以将上述参数设置为 0。另外这个参数还设置了这些周期性数据保存的最小时间周期，因此如果使用控制文件来存放 RMAN 的 CATALOG 信息，那么在设置这个参数时要十分注意。必须设置足够大的参数，以保证所有的 RMAN 恢复所需要的信息都是完整的。如果这个参数设置为 5，而一次数据库全备的时间间隔为 7 天，那么在做数据库恢复的时候，就可能找不到关于上一次全备的信息，导致恢复失败。

控制文件扩展是一种十分危险的操作，需要通过 CF 锁排斥其他的控制文件操作，直到扩展结束。如果这时相关的会话失效或者数据库实例宕机，那么控制文件就有可能被破坏，这种情况下，只能通过备份的控制文件来恢复了。从这一点也可以看出控制文件备份的重要性，另外要尽可能地防止控制文件的自动扩展，因此在创建数据库时，设定足够大的 DB_FILES 等参数是十分必要的。虽然控制文件能自动扩展，但我们应该尽可能事先通过参数的控制来减少这种扩展，以达到最高的安全性。

4.1.3 如何转储和分析控制文件

如果要研究控制文件，就需要转储控制文件，Oracle 提供了一个事件 CONTROLF 来实现控制文件的转储。这个命令很简单，我们可以用 oradebug 来操作转储。

```
sqlplus '/as sysdba'
SQL>oradebug setmypid;
SQL>oradebug dump controlf 3;
```

这样操作，一个转储文件就会生成在用户 dump 目录了。如果不习惯使用 oradebug，也可以使用 alter session 命令来执行：

```
alter session set events 'immediate trace name controlf level 3';
```

这里，老白使用了级别为 3 的转储，关于级别的描述见表 4-1。

表 4-1

Dump Level	Dump 内容
1	仅仅文件头
2	文件头、数据库信息记录，ckpt进程记录
3	所有的记录类型，不过对于循环的记录类型，仅仅转储最早和最后的记录
4	同上，再加上循环记录类型的最新4条记录
5+	同上，不过转储更多的循环记录类型记录

下面通过一个示例来介绍控制文件中包含的内容。

```
***************************************************************************
DATABASE ENTRY
***************************************************************************
 (size = 316, compat size = 316, section max = 1, section in-use = 1,
  last-recid= 0, old-recno = 0, last-recno = 0)
 (extent = 1, blkno = 1, numrecs = 1)
 09/26/2011 15:24:40
 DB Name "ORCL"
 Database flags = 0x00404000 0x00001000
 Controlfile Creation Timestamp  09/26/2011 15:24:40
 Incmplt recovery scn: 0x0000.00000000
 Resetlogs scn: 0x0000.000716f7 Resetlogs Timestamp  09/26/2011 15:25:38
 Prior resetlogs scn: 0x0000.00000001 Prior resetlogs Timestamp  02/17/2008 01:50:35
 Redo Version: compatible=0xa200400
 #Data files = 4, #Online files = 4
 Database checkpoint: Thread=1 scn: 0x0000.008ef261
 Threads: #Enabled=1, #Open=1, Head=1, Tail=1
 Max log members = 3, Max data members = 1
 Arch list: Head=0, Tail=0, Force scn: 0x0000.008e2e00scn: 0x0000.000716f7
 Activation ID: 1290675138
 Controlfile Checkpointed at scn:  0x0000.008f79dc 04/09/2012 13:13:35
 thread:0 rba:(0x0.0.0)
```

上面是数据库信息小节，这部分的大小是 316B，包含了 DB NAME 等信息。下面的小节是 CHECKPOINT 的信息。

```
***************************************************************************
CHECKPOINT PROGRESS RECORDS
***************************************************************************
 (size = 8180, compat size = 8180, section max = 11, section in-use = 0,
  last-recid= 0, old-recno = 0, last-recno = 0)
 (extent = 1, blkno = 2, numrecs = 11)
THREAD #1 - status:0x2 flags:0x0 dirty:0
low cache rba:(0xffffffff.ffffffff.ffff) on disk rba:(0x1c3.118fa.0)
on disk scn: 0x0000.008f9f3e 03/09/2012 19:30:56
resetlogs scn: 0x0000.000716f7 09/26/2011 15:25:38
heartbeat: 779018961 mount id: 1306171311
THREAD #2 - status:0x0 flags:0x0 dirty:0
low cache rba:(0x0.0.0) on disk rba:(0x0.0.0)
on disk scn: 0x0000.00000000 01/01/1988 00:00:00
resetlogs scn: 0x0000.00000000 01/01/1988 00:00:00
heartbeat: 0 mount id: 0
```

默认情况下，这一节包含 8 个 THREAD 的 CHECKPOINT 信息小节，合计 8180B。为了节省篇幅，老白只列出了其中两个。由于本数据库是单实例的，所以 THREAD#2 的信息都是 0。CHECKPOINT 的信息包含 low cache rba、on disk rba 和 on disk scn、resetlogs scn 等信息。可以看出，这个数据库 2011 年 9 月 26 日做过 resetlogs，现在已经写盘的数据是 2012 年 3 月 9 日 19 点 30 分 56 秒的数据。当 CHECKPOINT 发生的时候，这些数据会被更新。

由于篇幅所限，余下的 TRACE 文件老白就不一一介绍了，有兴趣的朋友可以尝试自己转储一个控制文件，转储文件比较容易理解。

4.1.4 文件头和控制文件信息

大家都知道，控制文件中包含文件的信息，比如，包含 FILE#=1 文件的信息：

```
DATA FILE #1:
  (name #7) /opt/oracle/oradata/orcl/system01.dbf
 creation size=0 block size=8192 status=0xe head=7 tail=7 dup=1
 tablespace 0, index=1 krfil=1 prev_file=0
 unrecoverable scn: 0x0000.00000000 01/01/1988 00:00:00
 Checkpoint cnt:541 scn: 0x0000.008ef261 04/08/2012 15:53:12
 Stop scn: 0xffff.ffffffff 03/20/2012 07:26:24
 Creation Checkpointed at scn:  0x0000.00000007 02/17/2008 01:50:54
 thread:0 rba:(0x0.0.0)
 Offline scn: 0x0000.000716f6 prev_range: 0
 Online Checkpointed at scn:  0x0000.000716f7 09/26/2011 15:25:38
 thread:1 rba:(0x1.2.0)
 Hot Backup end marker scn: 0x0000.00000000
 aux_file is NOT DEFINED
```

这部分信息可以通过 V$DATAFILE 视图查看到。实际上这个视图的信息也是从控制文件中获取到的，只要数据库处于 MOUNT 状态，控制文件打开后就可以获取这个视图的信息。

在数据库中还有一个视图——V$DATAFILE_HEADER，和 V$DATAFILE 不同的是，这个视图是从数据文件的文件头中提取的，能够反映出数据文件的情况。一般来说，这两个视图中的 CHECKPOINT_CHANGE# 应该是基本上一致的，如果我们发现 V$DATAFILE 中的 CHECKPOINT_CHANGE#大于 V$DATAFILE_HEADER 中的，那么说明数据文件需要恢复。

如果要从备份集中恢复数据库，一般来说，恢复过来的数据文件的文件头中，SCN 都会小于控制文件中记录的文件的 SCN（这是因为数据文件的备份往往早于控制文件的备份），完成数据文件的恢复后，可以利用归档日志将数据文件前滚到控制文件中所记录的 SCN 的位置。如果准备做非完全恢复，那么我们可能会恢复到某个 SCN，这种情况下，如何判断此时 OPEN DATABASE RESETLOGS 是否会成功呢？通过检查 V$DATAFILE_HEADER 就可以找到答案：

```
SQL> set line 200
SQL> select fuzzy, status, error, recover, checkpoint_change#, checkpoint_time,
count(*)
  2  from v$datafile_header
  3  group by fuzzy, status, error, recover, checkpoint_change#, checkpoint_time ;

FUZ STATUS  ERROR   REC CHECKPOINT_CHANGE# CHECKPOIN   COUNT(*)
--- ------- ------- --- ------------------ --------- ----------
 NO ONLINE                         9529285 09-APR-12          4
```

如上例所示，所有的文件的 FUZZY 都是 NO，说明此时 OPEN DATABASE RESETLOGS 是安全的。如果查询得到的结果与下面的信息类似：

```
FUZ STATUS  ERROR  REC CHECKPOINT_CHANGE# CHECKPOINT_   COUNT(*)
--- ------- ------ --- ------------------ ----------- ----------
NO  ONLINE                        5311260 31-AUG-2011          6
YES ONLINE                        5311260 31-AUG-2011          1
```

那么说明有一个文件是不一致的，可能需要多次恢复才能达到一致性的点。接下来，我们可以通过下面的 SQL 找出处于 FUZZY 状态的文件：

```
Select name from v$datafile_header where fuzzy='YES'
```

如果这个文件不是一般的数据文件，而是系统表空间，那么该数据库可能处于不一致状态，重置日志会失败，因此需要继续恢复下一个日志。如果不太走运，下一个日志已经丢失了，那么可能就无法通过正常的 OPEN RESETLOGS 打开这个数据库。这时通过使用隐含参数也许可以强制打开数据库，但是可能会引发一些其他问题，比如，出现 ORA-600[4XXX]或者 ORA-600[2662]之类的错误。关于这方面的处理方法，老白在其他章节已有所介绍，这里就不多讨论了。

4.2　故障处理和优化

控制文件的故障主要集中在文件丢失、损坏方面，另外，CF 锁冲突可能导致数据库出现挂起等问题，这也是需要 DBA 密切关注的。

4.2.1　丢失或者损坏控制文件的处理方法

控制文件损坏可能导致数据库实例故障，因此 DBA 必须掌握控制文件损坏的处理方法。单个控制文件损坏是比较容易恢复的，因为一般的数据库系统，控制文件都不止一个，而且所有的控制文件都互为镜像，只要复制一个完好的控制文件替换已损坏的控制文件就可以了。控制文件损坏所导致的最典型的问题就是启动数据库出错，不能加载数据库。

```
SQL>startup
ORA-00205: error in identifying controlfile, check alert log for more info
```

查看报警日志文件，有如下信息：

```
alter database mount
Mon May 14 13:59:51 2010
ORA-00202: controlfile: 'D:\oracle\oradata\orcl\control01.ctl'
ORA-27041: unable to open file
OSD-04002: unable to open file
O/S-Error: (OS 2) 系统找不到指定的文件。
```

遇到这种情况，需要首先关闭数据库实例。

```
SQL>shutdown abort
```

由于数据库根本就没有加载，所以 shutdown abort 是十分安全的。关闭数据库实例后，需要复制一个完好的控制文件替换已损坏的控制文件，或修改 init.ora 中的控制文件参数，取消已损坏的控制文件，然后重启数据库。

恢复单个控制文件是比较简单的，因为数据库中所有的控制文件都是镜像的，只需要简单地复制、替换就可以了。这也是我们建议在不同磁盘上镜像控制文件的主要原因，如果磁盘故障导致了某个控制文件损坏，只要还有一个控制文件是完好的，就可以进行恢复。

如果所有的控制文件都损坏了怎么办呢？这种情况下，就需要使用备份的控制文件了。如果

比较幸运，系统有备份，那么只要将备份取出，就可以恢复了。基于这样的原因，老白建议多做控制文件的备份，长期保留一份由 ALTER DATABASE BACKUP CONTROL FILE TO TRACE 产生的控制文件的文本备份（每次新增数据文件后都重新备份一次）。

上面我们讨论的是一些比较理想的场景。在某些情况下，我们可能不够走运，当有多个控制文件损坏了，或者人为删除了所有的控制文件，通过控制文件的复制已经不能解决问题时，就需要重新建立控制文件。同时应注意，ALTER DATABASE BACKUP CONTROL FILE TO TRACE 可以产生一个控制文件的文本备份。

以下是重新创建控制文件的详细步骤。

(1) 关闭数据库。

```
SQL>shutdown immediate;
```

(2) 删除所有控制文件，模拟控制文件丢失。

(3) 启动数据库，出现错误，且不能启动到加载状态下。

```
SQL>startup
ORA-00205: error in identifying controlfile, check alert log for more info
```

查看报警日志文件，有如下信息：

```
SQL>alter database mount
  Mon May 26 11:53:15 2003
  ORA-00202: controlfile: 'D:Oracleoradatachencontrol01.ctl'
  ORA-27041: unable to open file
  OSD-04002: unable to open file
  O/S-Error: (OS 2) 系统找不到指定的文件。
```

(4) 关闭数据库。

```
SQL>shutdown immediate;
```

(5) 在 internal 或 sys 下运行创建控制文件的脚本，注意完整地列出联机日志及数据文件的路径。此外，也可以修改由 ALTER DATABASE BACKUP CONTROL FILE TO TRACE 备份控制文件时产生的脚本，去掉多余的注释即可。

```
STARTUP NOMOUNT
    CREATE CONTROLFILE REUSE DATABASE "TEST" NORESETLOGS NOARCHIVELOG
    MAXLOGFILES 32
    MAXLOGMEMBERS 2
    MAXDATAFILES 254
    MAXINSTANCES 1
    MAXLOGHISTORY 226
    LOGFILE
    GROUP 1 'D:ORACLEORADATATESTREDO01.LOG' SIZE 1M,
    GROUP 2 'D:ORACLEORADATATESTREDO02.LOG' SIZE 1M,
    GROUP 3 'D:ORACLEORADATATESTREDO03.LOG' SIZE 1M
    DATAFILE
    'D:ORACLEORADATATESTSYSTEM01.DBF',
    'D:ORACLEORADATATESTRBS01.DBF',
    'D:ORACLEORADATATESTUSERS01.DBF',
```

```
'D:ORACLEORADATATESTTEMP01.DBF',
'D:ORACLEORADATATESTTOOLS01.DBF',
'D:ORACLEORADATATESTINDX01.DBF'
CHARACTER SET ZHS16GBK;
-- Recovery is required if any of the datafiles are restored backups,
-- or if the last shutdown was not normal or immediate.
RECOVER DATABASE
--if the last shutdown was not normal or immediate
--noarchive
-- RECOVER DATABASE UNTIL CANCEL USING BACKUP CONTROLFILE
--archive
-- RECOVER DATABASE USING BACKUP CONTROLFILE UNTIL CANCEL
-- Database can now be opened normally.
ALTER DATABASE OPEN;
--if recover database until cancel
--ALTER DATABASE OPEN RESETLOGS;
```

(6) 如果没有错误，数据库将启动到打开状态下。

上面介绍了重建控制文件的方法。重建控制文件用于恢复全部控制文件的损坏，需要注意其书写的正确性，保证包含了所有的数据文件与联机日志。我们在启动数据库之前，一定要确保所有的数据文件都已经被包含在重建的控制文件脚本中了。如果在编辑过程中误删了某个文件，当数据库打开后，要想再将其加入到数据库中，就需要恢复这个数据文件，否则无法使该数据文件处于在线状态。但如果丢失了归档日志，这个文件可能就无法再次加入到数据库中了。

在做数据库恢复时，我们经常会碰到这样一种情况：因为某个磁盘损坏了，不能再恢复或存储数据文件到这个磁盘，而在存储到其他磁盘时，就必须重新创建控制文件，用于识别这个新的数据文件。这时也可以用上述方法进行恢复。

下面来看一个更为极端的情况：丢失了所有的控制文件及备份控制文件，同时没有保存记录文件。这种情况下该如何处理呢？比较麻烦，我们需要根据系统中的文件或者裸设备，手工编写创建控制文件的命令。当然，也可以从其他类似的系统中复制一份文件来改写。只要足够仔细，没有遗漏任何文件，也可以达到目的。无论我们是否采用这种最为极端的方式来解决问题，老白都希望大家把工作做在前面，尽可能避免以这种方式来进行恢复。

4.2.2　控制文件的优化

在一般情况下，控制文件的变更较小，不会对系统的性能产生很大影响。由于控制文件的 I/O 量不大，并发访问量也不会很大，所以由于 I/O 性能问题导致控制文件出现问题的可能性十分小。控制文件的性能问题往往体现在控制文件锁 CF 方面。如果在某些情况下我们发现 CF 锁等待十分严重，那么可能就是控制文件出现了问题。

如果 CF 锁等待十分严重，很多情况下会导致数据库实例宕掉，或者需要重启实例才能彻底解决问题。导致 CF 锁的问题往往不是控制文件本身，因此要防止 CF 锁出现问题，就要从 CF 锁的使用者来考虑。一般来说，可能导致 CF 锁冲突的原因主要是 REDO LOG 切换频率过快、CKPT 过于频繁或归档日志出现故障。因此还是要从这几个角度去考虑预防 CF 锁冲突的方法，

而不能仅仅从控制文件本身去考虑。使用适当的 DB CACHE（不要过小）、确保 LOG FILE SYNC 响应时间、避免过于频繁的日志切换、确保 DATAGUARD 接收处于正常状态等都是较为有效的方法。

另外，控制文件扩展时也会持有 CF 锁。创建数据库时要设置足够大的参数，比如 DB_FILES 等，这样可以尽可能减少控制文件的扩展，进而减少控制文件扩展导致的问题。

第 5 章

理解 REDO 日志

REDO 日志文件是 Oracle 数据库中十分重要的文件，它记录了 Oracle 的所有变化，是 Oracle 数据库能够正常运行、不丢失数据的最根本保证。

REDO 日志文件的安全性是首先要得到保证的，因为一旦 REDO 日志文件出现故障或者丢失，可能导致数据丢失，数据库出现不一致甚至无法打开等重大事故。另外，REDO 日志是 Oracle 数据库中访问最频繁、写 IO 最多的文件，因此其访问性能对于数据库整体性能提升影响很大。

本节将从介绍 REDO 日志内部结构开始，和大家一起讨论 REDO 日志相关的一些维护、优化问题。

5.1 什么是 REDO 日志

REDO 日志文件是 Oracle 数据库实例恢复机制中最为关键的组成部分。REDO 日志机制的目的就是确保数据库实例或者服务器发生故障时，不会导致数据库崩溃，不会丢失已经提交的数据。Oracle 为了实现这个目标，将数据库的变化数据记录在 REDO 日志文件中，而且通过 Redo-Write-Ahead（RWA）机制确保数据库被变更之前，其变更的 REDO 日志信息必须先写入日志缓冲区，而事务提交之前也必须首先将日志缓冲中和这个事务相关的 REDO 信息写入 REDO 日志文件。

采用这个机制，Oracle 就可以确保在数据库宕机后，只要重新启动实例，数据库中没有被及时写入数据文件的 DB Cache 信息和一些不一致的未提交的事务信息会被正确恢复，而且数据库可以恢复到宕机前的一致性状态。

REDO 日志被设计为多个组，一个数据库可以包含多个 REDO 日志组。这样设计的好处是，当一组 REDO 日志完后，可以切换到另外一组上面继续写，因此不需要马上清除当前 REDO 日志中的信息。因为由于某些数据块的写盘操作还没完成，这些信息有可能还不能马上清除，否则数据库宕机后就没法完全恢复了。对于多实例的数据库来说，每个实例都有自己独立的 REDO 日志组，这样的设计确保了多实例数据库 REDO 日志的写性能，因为不同实例不需要并发写入同一组 REDO 日志，从而避免了争用。不过这种设计使得实例恢复更为复杂，因为所有实例的 REDO 日志中的变化（CHANGE#）都是严格排序的，并且是不会跳跃的，实例恢复所需要的变化可能包含在多个 REDO 日志文件中，因此恢复时无法像单实例那样只根据 REDO 日志的顺序往下处理，

而要不停地在多个实例的 REDO 日志中挑选所需要的变化，从而顺序地应用所有变化。

REDO 日志文件是十分关键的，因此和控制文件一样，Oracle 也支持对 REDO 日志文件进行镜像，我们可以设置一个日志组的文件数量为 2 或者更多。不过一般来说，设置为 2 就足够了，太多的文件会增加大量的 I/O，影响系统性能。使用 REDO 日志镜像时，最好能够将不同的文件放置在不同的 VG 上，甚至不同的磁盘组中，当然放置在不同的存储区是最安全的。如果我们使用 ASM，那么把这些文件放到不同的磁盘组中是很好的选择。Oracle 建议将两个文件中的一个放在数据磁盘组中，另一个放在闪回磁盘组（Flashback Diskgroup）中。如果将两个 REDO 日志文件都放在同一个 VG 或者文件系统中，虽然也能起到一定的容错作用，但是如果 VG 故障或者文件系统故障，我们将失去所有的文件，从而导致一些不可恢复的故障出现。

5.2　REDO 的基本原理

了解 REDO 日志的基本原理是很多 DBA 的心愿。虽然关于 REDO 日志内部原理的资料很多，但都过于零散，而且 REDO 日志本身十分复杂，想要深入了解其内部算法确实比较困难。老白和大家一样，只能依靠从零散的技术资料中获取的信息进行分析和学习。本节的主要目的就是总结这些学习成果，并分享给大家。

5.2.1　介质恢复和实例恢复的基本概念

REDO 日志是 Oracle 为确保已经提交的事务不会丢失而建立的一种机制。实际上，REDO 日志的存在是为两种场景准备的，一种我们称为实例恢复（Instance Recovery），另一种称为介质恢复（Media Recovery）。实例恢复的目的是在数据库发生故障时，确保数据块缓冲区中的数据不会丢失，不会造成数据库的不一致。介质恢复的目的是当数据文件发生故障时，能够恢复数据。虽然这两种恢复使用的机制类似，但是存在本质的不同，这一点也是很多 DBA 经常会混淆的。

REDO 日志的数据是按照线程来组织的。对于单实例系统来说，只有一个线程，而对于 RAC 系统来说，可能存在多个线程，每个数据库实例拥有一组独立的 REDO 日志文件，以及独立的日志缓冲区，某个实例的变化会被独立记录到一个线程的 REDO 日志文件中。

对于介质恢复和实例恢复来说，第一个步骤都是通过 REDO 日志的信息进行前滚。在做前滚时，通过 REDO 日志文件里记录的数据库变化矢量（稍后我们会详细地介绍数据库变化矢量 CV），根据 SCN 的比对，提交到相关的数据文件上，从而使数据文件的状态向前滚动。大家要注意的是，UNDO 表空间的变化也被记录到 REDO 日志里了，因此 UNDO 表空间相关的数据文件也会被前滚。当前滚到最后一个可用的 REDO 日志或者归档日志时，所有的数据库恢复层面的工作就全部完成了。这个时候，数据库包含了所有的被记录的变化，这些变化中有些是已经提交的，而有些是尚未提交的。在最新状态的 UNDO 表空间中，我们也可以看到一些尚未提交的事务。

因此数据库下一步需要做的就是事务层面的处理，回滚那些尚未提交的事务，以确保数据库的一致性。

对于单实例的系统，实例恢复一般是在数据库实例异常故障后数据库重启时进行，当数据库执行了 SHUTDOWN ABORT 命令或者由于操作系统、主机等原因宕机重启后，在 ALTER DATABASE OPEN 时，就会自动进行实例恢复。而在 RAC 环境中，如果某个实例宕掉了，活着的实例将会接管，替宕掉的实例做实例恢复。除非是所有的实例都宕掉了，这样的话，第一个执行 ALTER DATABASE OPEN 的实例将会做实例恢复。这也是 REDO 日志文件作为实例私有的组件必须存放在共享存储上的原因。

Oracle 数据库的高速缓存机制是以性能为导向的。高速缓存机制应该最大限度地提高数据库的性能，因此缓存被写入数据文件时总是尽可能推迟。这种机制大大提高了数据库的性能，但是当实例出现故障时，可能存在一些问题。

首先，可能某些事物对数据文件的修改并没有完全写入磁盘，或者磁盘文件中丢失了某些已提交事务对数据文件的修改信息。其次，可能某些还没有提交的事务对数据文件的修改已经被写入磁盘文件了。也有可能某个原子变更有一部分数据已经被写入文件，而另外一部分数据还没有被写入磁盘文件。实例恢复就是要通过 ONLINE REDO LOG 文件中记录的信息，自动完成上述数据的修复工作。这个过程是完全自动的，不需要人工干预。

在这个机制里，有两个问题需要解决。第一个是如何确保已经提交的事务不会丢失，第二个是如何在数据库性能和实例恢复所需要的时间上做出平衡，既确保数据库性能不会下降，又保证实例恢复可以快速进行。

解决第一个问题比较简单。Oracle 有一个机制，叫做 Log-Force-at-Commit，就是说，在事务提交时，和这个事务相关的 REDO 日志数据，包括 COMMIT 记录，都必须从日志缓冲区中写入 REDO 日志文件，此时事务提交成功的信号才能发送给用户进程。通过这种机制，哪怕已经提交的事务中的部分缓冲缓存还没有被写入数据文件就发生了实例故障，在做实例恢复的时候，也可以通过 REDO 日志的信息将不一致的数据前滚。

Oracle 是通过 CHECKPOINT 机制来解决第二个问题的。在 Oracle 数据库中，对缓冲缓存的修改操作是由前台进程完成的，但是前台进程只负责将数据块从数据文件中读到缓冲区中，不负责将缓冲区中修改过的数据写入数据文件。缓冲区写入数据文件的操作是由后台进程 DBWR 来完成的。DBWR 可以根据系统的负载情况以及数据块是否被其他进程使用，来将一部分数据块回写到数据文件中。在这种机制下，某个数据块被写回文件的时间可能具有一定的随机性，有些先修改的数据块可能比较晚才被写入数据文件。而 CHECKPOINT 机制就是对上述机制的一种有效补充。CHECKPOINT 发生的时候，CKPT 进程会要求 DBWR 进程将某个 SCN 以前的所有被修改的块都写回数据文件。这样，一旦这次 CHECKPOINT 完成，这个 SCN 前的所有数据变更都已经存盘。如果之后发生了实例故障，在进行实例恢复时，只需要从这次 CHECKPOINT 已经完成后的变化量开始就行了，CHECKPOINT 之前的变化就不需要再去考虑了。

到目前为止，我们了解了实例恢复机制的一些基本原理，就可以大体上理解 REDO 日志的工作机制了。不过我们还需要更加深入一些，了解一些内幕。通过上面的内容，大家也许觉得对实例恢复已经了解得很透彻了，但实际上，还有很多问题没有解决。有些爱动脑筋的读者可能要问了，如果数据文件中的变化已经写盘，但是 REDO 日志信息还在日志缓冲区中，没有写入 REDO

日志，这种情况如何恢复呢?

这里我们又要引入一个名词 Write-Ahead-Log，就是日志写入优先。日志写入优先包含两方面的算法。第一个方面是，在某个缓冲区缓存的修改的变化矢量还没有写入 REDO 日志文件之前，这个修改后的缓冲区的数据不允许被写入数据文件，这样就确保了在数据文件中不可能包含未在 REDO 日志文件中记录的变化。第二个方面是，在对某个数据的 UNDO 信息的变化矢量没有被写入 REDO 日志之前，这个缓冲区的修改不能被写入数据文件。

介质恢复和实例恢复的机制是类似的，所不同的是，介质恢复是在存储的数据文件出现故障时进行的，且无法自动进行，必须手工执行 recover database 或者 recover datafile 命令来实施。一般来说，介质恢复是以一个恢复的数据文件为起点进行恢复的，因此在进行介质恢复时，需要使用归档日志。

5.2.2　变化矢量和 REDO 记录

本节将介绍一些 REDO 日志底层的概念。只有明白了这些概念，我们才能更加深入地了解 REDO 日志及其相关的管理和优化要点。首先我们要了解的就是变化矢量（Change Vector，CV）。变化矢量是组成 REDO 信息的基础，一个变化矢量描述了对一个独立数据块进行的一次独立修改操作。这里要注意的是，CV 的定义里包含了两层含义，即一个 CV 只针对一个数据块的变更，一个 CV 只包含一个变化。每个 CV 都包含了对文件的修改，因此在每个 CV 中都有一个 OPCODE 指出修改的类型。不同 OPCODE 的 CV，其组成也是不同的，OPCODE 的取值范围如代码清单 5-1 所示。

代码清单 5-1

```
Layer 1 : Transaction Control - KCOCOTCT
     Opcode 1 : KTZFMT
     Opcode 2 : KTZRDH
     Opcode 3 : KTZARC
     Opcode 4 : KTZREP

Layer 2 : Transaction Read -  KCOCOTRD

Layer 3 : Transaction Update -  KCOCOTUP

Layer 4 : Transaction Block -  KCOCOTBK     [ktbcts.h]
     Opcode 1 : Block Cleanout
     Opcode 2 : Physical Cleanout
     Opcode 3 : Single Array Change
     Opcode 4 : Multiple Changes to an Array
     Opcode 5 : Format Block

Layer 5 : Transaction Undo -  KCOCOTUN     [ktucts.h]
     Opcode 1 : Undo block or undo segment header - KTURDB
     Opcode 2 : Update rollback segment header - KTURDH
     Opcode 3 : Rollout a transaction begin
     Opcode 4 : Commit transaction (transaction table update)
```

```
      - no undo record
   Opcode 5 : Create rollback segment (format) - no undo record
   Opcode 6 : Rollback record index in an undo block - KTUIRB
   Opcode 7 : Begin transaction (transaction table update)
   Opcode 8 : Mark transaction as dead
   Opcode 9 : Undo routine to rollback the extend of a rollback segment
   Opcode 10 : Redo to perform the rollback of extend of rollback segment
               to the segment header.
   Opcode 11 : Rollback DBA in transaction table entry - KTUBRB
   Opcode 12 : Change transaction state (in transaction table entry)
   Opcode 13 : Convert rollback segment format (V6 -> V7)
   Opcode 14 : Change extent allocation parameters in a rollback segment
   Opcode 15 :
   Opcode 16 :
   Opcode 17 :
   Opcode 18 :
   Opcode 19 : Transaction start audit log record
   Opcode 20 : Transaction continue audit log record
   Opcode 24 : Kernel Transaction Undo Relog CHanGe - KTURLGU

Layer 6 : Control File -  KCOCODCF     [tbs.h]

Layer 10 : INDEX -  KCOCODIX     [kdi.h]
   Opcode 1 : load index block (Loader with direct mode)
   Opcode 2 : Insert leaf row
   Opcode 3 : Purge leaf row
   Opcode 4 : Mark leaf row deleted
   Opcode 5 : Restore leaf row (clear leaf delete flags)
   Opcode 6 : Lock index block
   Opcode 7 : Unlock index block
   Opcode 8 : Initialize new leaf block
   Opcode 9 : Apply Itl Redo
   Opcode 10 : Set leaf block next link
   Opcode 11 : Set leaf block previous link
   Opcode 12 : Init root block after split
   Opcode 13 : Make leaf block empty
   Opcode 14 : Restore block before image
   Opcode 15 : Branch block row insert
   Opcode 16 : Branch block row purge
   Opcode 17 : Initialize new branch block
   Opcode 18 : Update keydata in row
   Opcode 19 : Clear row's split flag
   Opcode 20 : Set row's split flag
   Opcode 21 : General undo above the cache (undo)
   Opcode 22 : Undo operation on leaf key above the cache (undo)
   Opcode 23 : Restore block to b-tree
   Opcode 24 : Shrink ITL (transaction entries)
   Opcode 25 : Format root block redo
   Opcode 26 : Undo of format root block (undo)
   Opcode 27 : Redo for undo of format root block
   Opcode 28 : Undo for migrating block
   Opcode 29 : Redo for migrating block
   Opcode 30 : IOT leaf block nonkey update
   Opcode 31 : Cirect load root redo
   Opcode 32 : Combine operation for insert and restore rows
```

```
Layer 11 : Row Access -  KCOCODRW     [kdocts.h]
     Opcode 1 : Interpret Undo Record (Undo)
     Opcode 2 : Insert Row Piece
     Opcode 3 : Drop Row Piece
     Opcode 4 : Lock Row Piece
     Opcode 5 : Update Row Piece
     Opcode 6 : Overwrite Row Piece
     Opcode 7 : Manipulate First Column (add or delete the 1rst column)
     Opcode 8 : Change Forwarding address
     Opcode 9 : Change the Cluster Key Index
     Opcode 10 : Set Key Links (change the forward & backward key links
                 on a cluster key)
     Opcode 11 : Quick Multi-Insert (ex: insert as select ...)
     Opcode 12 : Quick Multi-Delete
     Opcode 13 : Toggle Block Header flags

Layer 12 : Cluster -  KCOCODCL     [?]

Layer 13 : Transaction Segment -  KCOCOTSG     [ktscts.h]
     Opcode 1 : Data segment format
     Opcode 2 : Merge
     Opcode 3 : Set link in block
     Opcode 4 : Not used
     Opcode 5 : New block (affects segment header)
     Opcode 6 : Format block (affects data block)
     Opcode 7 : Record link
     Opcode 8 : Undo free list (undo)
     Opcode 9 : Redo free list head (called as part of undo)
     Opcode 9 : Format free list block (freelist group)
     Opcode 11 : Format new blocks in free list
     Opcode 12 : free list clear
     Opcode 13 : free list restore (back) (undo of opcode 12)

Layer 14 : Transaction Extent -  KCOCOTEX     [kte.h]
     Opcode 1 : Add extent to segment
     Opcode 2 : Unlock Segment Header
     Opcode 3 : Extent DEaLlocation (DEL)
     Opcode 4 : Undo to Add extent operation (see opcode 1)
     Opcode 5 : Extent Incarnation number increment
     Opcode 6 : Lock segment Header
     Opcode 7 : Undo to rollback extent deallocation (see opcode 3)
     Opcode 8 : Apply Position Update (truncate)
     Opcode 9 : Link blocks to Freelist
     Opcode 10 : Unlink blocks from Freelist
     Opcode 11 : Undo to Apply Position Update (see opcode 8)
     Opcode 12 : Convert segment header to 6.2.x type

Layer 15 : Table Space -  KCOCOTTS     [ktt.h]
     Opcode 1 : Format deferred rollback segment header
     Opcode 2 : Add deferred rollback record
     Opcode 3 : Move to next block
     Opcode 4 : Point to next deferred rollback record

Layer 16 : Row Cache -  KCOCOQRC
```

```
Layer 17 : Recovery (REDO) -  KCOCORCV     [kcv.h]
     Opcode 1 : End Hot Backup : This operation clears the hot backup
                in-progress flags in the indicated list of files
     Opcode 2 : Enable Thread : This operation creates a redo record
                signalling that a thread has been enabled
     Opcode 3 : Crash Recovery Marker
     Opcode 4 : Resizeable datafiles
     Opcode 5 : Tablespace ONline
     Opcode 6 : Tablespace OFFline
     Opcode 7 : Tablespace ReaD Write
     Opcode 8 : Tablespace ReaD Only
     Opcode 9 : ADDing datafiles to database
     Opcode 10 : Tablespace DRoP
     Opcode 11 : Tablespace PitR

Layer 18 : Hot Backup Log Blocks -  KCOCOHLB     [kcb.h]
     Opcode 1 : Log block image
     Opcode 2 : Recovery testing

Layer 19 : Direct Loader Log Blocks - KCOCODLB     [kcbl.h]
     Opcode 1 : Direct block logging
     Opcode 2 : Invalidate range
     Opcode 3 : Direct block relogging
     Opcode 4 : Invalidate range relogging

Layer 20 : Compatibility Segment operations - KCOCOKCK  [kck.h]
     Opcode 1 : Format compatibility segment -  KCKFCS
     Opcode 2 : Update compatibility segment -  KCKUCS

Layer 21 : LOB segment operations - KCOCOLFS     [kdl2.h]
     Opcode 1 : Write data into ILOB data block - KDLOPWRI

Layer 22 : Tablespace bitmapped file operations -  KCOCOTBF [ktfb.h]
 Opcode 1 : format space header - KTFBHFO
 Opcode 2 : space header generic redo - KTFBHREDO
 Opcode 3 : space header undo - KTFBHUNDO
 Opcode 4 : space bitmap block format - KTFBBFO
 Opcode 5 : bitmap block generic redo - KTFBBREDO

Layer 23 : write behind logging of blocks - KCOCOLWR [kcbb.h]
 Opcode 1 : Dummy block written callback - KCBBLWR

Layer 24 : Logminer related (DDL or OBJV# redo) - KCOCOKRV [krv.h]
 Opcode : common portion of the ddl - KRVDDL
 Opcode : direct load redo - KRVDLR
 Opcode : lob related info - KRVLOB
 Opcode : misc info - KRVMISC
 Opcode : user info - KRVUSER
```

REDO 记录是由一组 CV 组成的，这组 CV 完成对数据库的一个原子修改操作。比如，一个 REDO 记录里可能包含 3 个 CV，第一个是对 UNDO SEGMENT HEADER 的修改，第二个是对 UNDO SEGMENT 的修改，第三个是对 DATA BLOCK 的修改。而一个事务可能包含多个 REDO 记录。

当前台进程要对某个数据块进行修改的时候，首先要形成相关的 CV，然后把多个 CV 组成 REDO 记录，再把 REDO 记录写入日志缓冲区后，前台进程可以将 CV 提交到相关的数据块上。

下面通过一个示例来学习 REDO 记录和变化矢量。我们设计的场景是，首先在 SCOTT 下创建一张表——SCOTT.T4：

```
create table t4 (a integer);
```

然后查看当前的 SCN 信息：

```
SQL> select max(ktuxescnw * power(2, 32) + ktuxescnb) from x$ktuxe;
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            16230857
```

执行一条 INSERT 语句，然后以这条语句进行分析：

```
insert into t4 values (1);
commit;
 SQL> select max(ktuxescnw * power(2, 32) + ktuxescnb) from x$ktuxe;
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            16230863
```

通过这两个 SCN 值，对当前的 REDO 日志进行转储：

```
SQL> alter system dump logfile 'd:\oracle\oradata\ora92\redo01.log' scn min 16230857
scn max 16230863;
系统已更改。
```

下面的内容就是 INSERT 语句所产生的，第一条 REDO 记录内容如下：

```
REDO RECORD - Thread:1 RBA: 0x0000a1.000040ce.0010 LEN: 0x0054 VLD: 0x01
SCN: 0x0000.00f7a9c9 SUBSCN:  1 03/12/2008 09:37:49
CHANGE #1 TYP:0 CLS:33 AFN:2 DBA:0x00800111 SCN:0x0000.00f7a9c7 SEQ:  1 OP:5.4
```

可以看出，这个 REDO 记录的 RBA 是 0x0000a1.000040ce.0010，转换成十进制就是 161.64.16，LOG SEQUENCE 号是 161，在 REDO 日志中的块号是 64，起始字节是块内的 16B。这个 REDO 记录的长度是 84B（0X54）。

“VLD:0X01”表示该 REDO 记录的类型，“0X01”表示 CHANGE VECTOR COPIED IN。“CHANGE #1”表示这是该 REDO 记录的第一个 CV。根据上面的 OPCODE 清单可知，“OP:5.4”表示 Commit transaction（transaction table update），即修改事务表。RDBA 是“2/273”，通过 DBA_EXTENTS 查询为_SYSSMU3$。

继续看下一条 REDO 记录：

```
REDO RECORD - Thread:1 RBA: 0x0000a1.000040cf.0010 LEN: 0x0058 VLD: 0x02
SCN: 0x0000.00f7a9cb SUBSCN:  1 03/12/2008 09:37:57
CHANGE #1 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:23.1
 Block Written - afn: 2 rdba: 0x008095f1(2,38385)                  ----undo segment
                    scn: 0x0000.00f79cf1 seq: 0x02 flg:0x04
 Block Written - afn: 2 rdba: 0x0080726b(2,29291)                ------undo segment
                    scn: 0x0000.00f7a1e1 seq: 0x02 flg:0x04      ------undo segment
 Block Written - afn: 2 rdba: 0x00806e2e(2,28206)                ------undo segment
                    scn: 0x0000.00f79d50 seq: 0x02 flg:0x04
```

这个 REDO 记录的 VLD 是 2，表示这个 REDO 记录里分配并存储了一个 SCN。来看第一个 CHANGE：

OP:23.1，即 OPCODE 为 23.1，表示产生一些 WRITE BEHIND LOGGING 信息（Dummy block written callback - KCBBLWR）。下一条 REDO 记录开始是针对 T4 表的操作。

```
REDO RECORD - Thread:1 RBA: 0x0000a1.000040d0.0010 LEN: 0x00b8 VLD: 0x01
SCN: 0x0000.00f7a9cd SUBSCN:  1 03/12/2008 09:38:03
CHANGE #1 TYP:0 CLS: 4 AFN:5 DBA:0x01401a63 SCN:0x0000.00f7a965 SEQ:  3 OP:13.28
Redo on Level1 Bitmap Block  ---针对 1ST BMB 的操作
Redo to add range
bdba: Length: 16
CHANGE #2 TYP:0 CLS: 8 AFN:5 DBA:0x01401a61 SCN:0x0000.00f7a965 SEQ:  2 OP:13.22
----dba (5/6753) -- scott.t4 的 segment header，设置高水位
Redo on Level1 Bitmap Block
Redo to set hwm
Opcode: 32      Highwater::  0x01401a71  ext#: 0      blk#: 16     ext size: 16
  #blocks in seg. hdr's freelists: 0
  #blocks below: 13
  mapblk  0x00000000  offset: 0
```

这个 REDO 记录是对 SCOTT.T4 表头的操作。下面的几个 REDO 记录都是格式化数据块，为了简化起见，只列出其中的一个：

```
REDO RECORD - Thread:1 RBA: 0x0000a1.000040d0.00c8 LEN: 0x003c VLD: 0x01
SCN: 0x0000.00f7a9cd SUBSCN:  1 03/12/2008 09:38:03
CHANGE #1 TYP:1 CLS: 1 AFN:5 DBA:0x01401a64 SCN:0x0000.00f7a9cd SEQ:  1 OP:13.21
--dba(5/6756) --scott.t4，格式化 BLOCK
ktspbfredo - Format Pagetable Datablock
Parent(l1) DBA: typ: 1 objd: 32027 itls: 2 fmt_flag: 0 poff: 0
```

中间省略了几个 REDO 记录，直接来看包含 INSERT 语句的 REDO 记录：

```
REDO RECORD - Thread:1 RBA: 0x0000a1.000040d2.00b0 LEN: 0x015c VLD: 0x01
SCN: 0x0000.00f7a9cd SUBSCN:  1 03/12/2008 09:38:03
CHANGE #1 TYP:0 CLS:23 AFN:2 DBA:0x00800071 SCN:0x0000.00f7a38b SEQ:  1 OP:5.2
----Update rollback segment header - KTURDH SYS_SYSSMU2$
ktudh redo: slt: 0x000f sqn: 0x00004947 flg: 0x0012 siz: 80 fbi: 0
            uba: 0x008090cb.0550.13    pxid:  0x0000.000.00000000
CHANGE #2 TYP:0 CLS:24 AFN:2 DBA:0x008090cb SCN:0x0000.00f7a38a SEQ:  3 OP:5.1
---Undo block or undo segment header - KTURDB  SYS_SYSSMU4$
ktudb redo: siz: 80 spc: 2746 flg: 0x0012 seq: 0x0550 rec: 0x13
            xid:  0x0004.00f.00004947
ktubl redo: slt: 15 rci: 0 opc: 11.1 objn: 32027 objd: 32027 tsn: 5
----Interpret Undo Record (Undo) ，针对 scott.t4 表生成 UNDO 数据
Undo type:  Regular undo        Begin trans    Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
             0x00000000  prev ctl uba: 0x008090cb.0550.10
prev ctl max cmt scn:  0x0000.00f78d7e  prev tx cmt scn:  0x0000.00f78f06
KDO undo record:
KTB Redo
op: 0x03  ver: 0x01
op: Z
------Undo of first (ever) change to the ITL，首先是对 ITL 的修改
```

```
KDO Op code: DRP row dependencies Disabled
------Delete Row Piece
  xtype: XA  bdba: 0x01401a65  hdba: 0x01401a63
-----ROWID
itli: 1  ispac: 0  maxfr: 2401
tabn: 0 slot: 0(0x0)
CHANGE #3 TYP:0 CLS: 1 AFN:5 DBA:0x01401a65 SCN:0x0000.00f7a9cd SEQ:  2 OP:11.2
---Insert Row Piece ，插入一条记录
KTB Redo
op: 0x01  ver: 0x01
op: F  xid:  0x0004.00f.00004947    uba: 0x008090cb.0550.13
---First change to ITL by this TX. Copy redo to ITL
KDO Op code: IRP row dependencies Disabled
---Single Insert Row Piece，行插入操作
  xtype: XA  bdba: 0x01401a65  hdba: 0x01401a63
---对应的表是 scott.t4
itli: 1  ispac: 0  maxfr: 2401
tabn: 0 slot: 0(0x0) size/delt: 6
fb: --H-FL-- lb: 0x1  cc: 1
null: -
col  0: [ 2]  c1 02
----十进制 1，就是我们插入的数据
CHANGE #4 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:5.20
----Transaction continue audit log record，记录 SESSION 信息
session number   = 11
serial  number   = 115
transaction name =
```

这个事务的最后一条 REDO 记录就是 COMMIT 产生的记录：

```
REDO RECORD - Thread:1 RBA: 0x0000a1.000040d4.0010 LEN: 0x0054 VLD: 0x01
SCN: 0x0000.00f7a9cf SUBSCN:  1 03/12/2008 09:38:03
CHANGE #1 TYP:0 CLS:23 AFN:2 DBA:0x00800071 SCN:0x0000.00f7a9cd SEQ:  1 OP:5.4
----Commit transaction (transaction table update)
ktucm redo: slt: 0x000f sqn: 0x00004947 srt: 0 sta: 9 flg: 0x2
ktucf redo: uba: 0x008090cb.0550.13 ext: 2 spc: 2664 fbi: 0
```

5.2.3　日志缓冲和 LGWR

REDO 日志的产生十分频繁，几乎每秒都有几十万字节到几兆字节的 REDO 日志产生，某些大型数据库每秒产生的 REDO 日志量甚至达到了 10MB 以上（老白曾见过一个每秒产生近 200MB REDO 日志的系统，不过这是压力测试环境下的数据）。不过前台进程每次产生的 REDO 量并不大，一般在几百字节到几千字节，而一般来说，一个事务所产生的 REDO 量也大致如此。基于 REDO 的这个特点，如果每次 REDO 产生后都必须写入 REDO 日志文件，那么就会存在两个问题。第一个是 REDO 日志文件写入的频率过高，会导致其 I/O 性能出现问题。第二个是如果由前台进程来完成 REDO 日志的写入，那么会导致大量并发的前台进程产生 REDO 日志文件的争用。为了解决这两个问题，Oracle 在 REDO 日志机制中引入了 LGWR 后台进程和日志缓冲。

日志缓冲用来缓存前台进程产生的 REDO 日志信息。前台进程可以将产生的 REDO 日志信息写入日志缓冲，而不需要直接写入 REDO 日志文件，这样就大大提高了 REDO 日志产生和保

存的时间，从而提高数据库在高并发情况下的性能。

既然前台进程不将 REDO 日志信息写入 REDO 日志文件了，那么就必须要有一个后台进程来完成这个工作。这个后台进程就是 LGWR，LGWR 进程的主要工作就是将日志缓冲中的数据批量写入到 REDO 日志文件中。在 Oracle 数据库中，只要数据库的改变被写入到 REDO 日志文件中，那么就可以确保相关的事务不会丢失了。

引入日志缓冲提高了整个数据库 RDBMS 写日志的性能，但是如何确保一个已经提交的事务确实被保存在数据库中，不会因为之后数据库发生故障而丢失呢？实际上，在前面两节中介绍的 REDO 日志的一些基本算法就可以确保这一点。首先，Write-Ahead-Log 协议确保了只要保存到 REDO 日志文件中的数据库变化一定能够被重演，不会丢失，也不会产生二义性。其次，在事务提交的时候，会产生一个提交 CV，这个 CV 被写入日志缓冲后，前台进程会发出一个信号，要求 LGWR 将和这个事务相关的 REDO 日志信息写入到 REDO 日志文件中。只有这个事务相关的 REDO 日志信息确实被写入 REDO 日志文件时，前台进程才会向客户端发出事务提交成功的消息，这样一个事务才算是被提交完成了。在这个协议下，只要客户端收到了提交完成的消息，那么就可以确保该事务已经存盘，不会丢失。LGWR 会绕过操作系统的缓冲，直接写入数据文件，以确保 REDO 日志的信息不会因为操作系统出现故障（比如宕机）而丢失应写入 REDO 日志文件的数据。

实际上，虽然 Oracle 数据库使用了绕过缓冲直接写 REDO 日志文件的方法，以避免操作系统故障导致的数据丢失，但我们还是无法确保这些数据已经被写到了物理磁盘上。因为 RDBMS 使用的绝大多数存储系统都是带有写缓冲的，写缓冲可以有效地提高存储系统写性能，不过也带来了另外的一个问题，就是一旦存储出现故障，可能会导致 REDO 日志的信息丢失，甚至导致 REDO 日志严重损坏。尽管存储故障出现的概率较小，但是这种小概率事件一旦发生，还是会导致一些数据库事务的丢失。因此，虽然 Oracle 的内部算法可以确保一旦事务提交成功，就被保存完毕，但是提交成功的事务还是可能丢失。

实际上，Oracle 在设计 REDO 日志文件的时候，已经最大限度地考虑了其安全性，REDO 日志文件的块大小和数据库完全不同，但和操作系统的 I/O 块大小完全相同。这种设计确保了一个 REDO 日志块在一次物理 I/O 中被同时写入，因此 REDO 日志块不会出现块断裂的现象。

了解日志缓冲和 LGWR 的算法，有助于我们分析和解决相关的性能问题。用一句话来概括——日志缓冲是一个循环使用的顺序型缓存。这里包含了两层含义。首先，日志缓冲是顺序读写的缓冲；其次，日志缓冲是循环缓冲，当日志缓冲写满后，会回到头部来继续写入 REDO 日志信息。日志缓冲数据的写入是由前台进程来完成的，并且是并发的，每个前台进程在生成了 REDO 日志信息后，需要首先在日志缓冲中分配空间，然后将 REDO 日志信息写入到日志缓冲中去。在日志缓冲中分配空间是一种串行的操作，因此 Oracle 在设计这方面的算法时，把日志缓冲空间分配和复制 REDO 日志数据到日志缓冲这两种操作分离了。一旦分配了日志缓冲空间，就可以释放相关的闩，这样，其他前台进程就可以继续分配空间了。（这里所说的前台进程是一种泛指，只是为了表述方便而已，读者一定要注意，因为后台进程也会对数据库进行修改，也需要产生 REDO 日志信息，后台进程的 REDO 操作和前台进程是大体一致的。）

前台进程写入 REDO 信息会使日志缓冲的尾部指针不停地向前推进，而 LGWR 这个后台进程会不停地从日志缓冲的头部指针开始查找还未写入 REDO 日志文件的日志缓冲信息，然后将这些信息写入 REDO 日志文件中，并将缓冲头部指针不停地向后推进。若日志缓冲的头部指针和尾部指针重合，就说明当前的日志缓冲是空的。而如果前台进程在日志缓冲中分配空间，会使日志缓冲的尾部指针一直向前推进。一旦日志缓冲的尾部指针追上了头部指针，日志缓冲就无法再分配新的空间给后台进程，后台进程必须要等候 LGWR 将这些数据写入 REDO 日志文件，然后向前推进头部指针，才可能再次获得新的可用缓冲空间。这时，前台进程会等待 LOG FILE SYNC 事件。

为了让 LGWR 尽快将日志缓冲中的数据写入 REDO 日志文件，以便于腾出更多的空闲空间，Oracle 数据库设计了 LGWR 写的触发条件。

- 事务提交时。
- 日志缓冲中的数据超过 1MB 时。
- 当日志缓冲中的数据超过了_log_io_size 隐含参数指定的大小时。
- 每隔 3 秒。

前面多次提到过，当事务提交时，会产生一个提交的 REDO 记录。这个记录写入日志缓冲后，前台进程会触发 LGWR 写操作。这时前台进程就会等待 LOG FILE SYNC 事件，直到 LGWR 将相关的数据写入 REDO 日志文件，等待才会结束，前台进程就会收到提交成功的消息。如果系统每秒的事务数量较大，比如几十个或者几百个，大型 OLTP 系统甚至会达到数千个，在这种系统中，LGWR 由于事务提交而被激发的频率很高，日志缓冲的信息会被很快地写入 REDO 日志文件。

而对于某些系统来说，如果每个事务的平均大小很大，生成的 REDO 日志平均数据量也很大，比如 1MB 甚至更高，每秒的平均事务数却很少，比如一两个甚至不到一个，那么这种系统中 LGWR 由于事务提交而被激发的频率很低。这样，就可能导致 REDO 日志信息在日志缓冲中被大量积压，日志缓冲中数据超过 1MB 的 LGWR 激发条件就是为了解决这种情况而设计的。当日志缓冲中的积压数据很多时，即便没有事务提交，也会触发 LGWR 将缓冲中的数据写入 REDO 日志文件。

除此之外，Oracle 还通过_log_io_size 这个隐含参数来进一步控制 LGWR 写操作，当日志缓冲中的数据超过了该参数规定的大小时，LGWR 就会被激发。这个参数的默认值是日志缓冲容量的 1/3，单位是 REDO LOG Block，可以控制当日志缓冲中有多少个数据块被占用时，就要触发 LGWR 写操作，从而避免日志缓冲被用尽。

如果一个空闲的系统很长时间都没有事务提交，日志缓冲的使用也很少，就可能导致日志缓冲中的数据长期没有被写入 REDO 日志文件，带来丢失数据的风险。因此 Oracle 还设计了一个激发 LGWR 写操作的条件，设置一个时间触发器，每隔 3 秒这个触发器都会被激活。被激活时，如果发现日志缓冲不为空，并且 LGWR 未处于活跃状态，就会产生一个事件，激活 LGWR。

前面我们讨论了 LGWR 和日志缓冲的一些基本算法，下面来介绍 LOG FILE SYNC 等待事件。该事件的作用是等待 LGWR 将日志缓冲的数据写入 REDO 日志文件。一般情况下，某个事

务在做提交时，会等待 LOG FILE SYNC 事件，而没有做提交操作的会话则不需要等待，因为前台进程只需要将 REDO 日志信息写入日志缓冲就可以了，不需要等待这些数据被写入 REDO 日志文件。不过前台进程在分配日志缓冲时，如果发现日志缓冲的尾部指针已经追上了头部指针，那么前台进程就要等待 LGWR 进程将头部的数据写入 REDO 日志文件,然后释放日志缓冲空间。这时，没有做提交操作的前台进程都会等待 LOG FILE SYNC 事件。这种情况下，加大日志缓冲就可以减少大部分的 LOG FILE SYNC 等待。

增加日志缓冲的大小，可能会带来另外一个问题，比如，日志缓冲从 1MB 增加到 30MB（关于日志缓冲是否需要大于 3MB 的问题，以前我们已经多次讨论，这里不再赘述，大家只需要记住一点就可以了，日志缓冲大于 3MB 是浪费空间、对性能影响不大的观点是错误的），那么_log_io_size 会自动从 300KB 增加到 10MB。在一个平均每秒事务数较少，并且每个事务的 REDO 尺寸较大的系统中，触发 LGWR 写操作的日志缓冲数据量会达到 1MB。一般来说，在一个大型的 OLTP 系统里，每次 LGWR 写入 REDO 日志文件的大小在几千字节到几万字节之间，LOG FILE SYNC 事件的平均等待时间在 1 ~ 10ms。如果平均每次写入的数据量过大，会导致 LOG FILE SYNC 的等待时间变长。因此在这种情况下，可能就需要设置_log_io_size 参数，确保 LOG FILE SYNC 等待不会过长。

如果每次写入 REDO 日志文件的数据量不大，而 LOG FILE SYNC 等待时间却很长，比如超过 100ms，那么就要分析一下 REDO 日志文件的 I/O 性能了。如果 REDO 日志文件 I/O 性能不佳，或者该文件所在的 I/O 热点较大，都可能导致 LOG FILE SYNC 等待时间偏大，这种情况下，我们可以查看后台进程的 LOG FILE PARALLEL WRITE 等待事件。这个事件一般的等待时间为几毫秒，如果该事件的平均等待时间较长，那么就说明 REDO 日志文件的 I/O 性能不佳，需要将 REDO 日志文件放到 I/O 量较小、性能较快的磁盘上。

在 OLTP 系统中，REDO 日志文件的写操作主要是小型的，比较频繁，一般大小为几千字节，而每秒钟产生的写 I/O 次数会达到几十、数百甚至上千。因此 REDO 日志文件适合存放于 IOPS 较高、转速较快的磁盘上，而 IOPS 仅能达到数百次的 SATA 盘不适合存放 REDO 日志文件。另外，REDO 日志文件的写入是串行的，因此对其所做的底层条带化处理，对于 REDO 日志写性能的提升是十分有限的。

5.2.4　日志切换和 REDO 日志文件

当前台进程在日志缓冲区中分配空间的时候，实际上已经在 REDO 日志文件中预先分配了空间。如果 REDO 日志文件已经写满，无法再分配空间给前台进程，就需要做一次日志切换。这时前台进程会向 LGWR 发出一个日志切换的请求，然后等待 LOG FILE SWITCH COMPLETION 事件。

日志切换请求发出后，CKPT 进程会进行一次日志切换 CHECKPOINT，而 LGWR 开始进行日志切换工作。首先 LGWR 进程会通过控制文件中的双向链表，查找一个可用的 REDO 日志文件，将其作为新的当前 REDO 日志。此查找算法要求该日志是非活动的，并且已经完成了归档（如果是归档模式）。Oracle 会优先使用空闲状态的 REDO 日志组作为当前 REDO 日志。

在做日志切换时，首先要将日志缓冲中还没有写入 REDO 日志文件的 REDO 记录写入当前 REDO 日志文件，然后将最后一个 REDO 记录的 SCN 作为该日志文件的 High SCN 记录在 REDO 日志文件头中。这些操作完成后，就可以关闭当前日志了。

完成了上一步骤后，就需要进行第二次控制文件事务，将刚刚关闭的 REDO 日志标识为 active，将新的当前 REDO LOG 标识为 current。如果数据库处于归档模式，还要将旧日志组记录到控制文件归档列表记录中（在 V$ARCHIVE 视图中可看到），并且通知归档进程对该日志文件进行归档。在所有的归档进程都处于忙碌状态，并且归档进程总数没有超过 log_archive_max_processes 的情况下，LGWR 还会生成一个新归档进程来对旧日志文件进行归档。

这些操作都完成后，LGWR 会打开新日志组的所有成员，并在文件头中记录下初始化信息。接下来，LGWR 将修改 SGA 中的标志位，允许生成新的 REDO 日志信息。

旧日志组目前仍被标志为 active，当 DBWR 完成了 CHECKPOINT 所要求的批量写操作后，该日志组的状态会被标识为 inactive。

从上述日志切换的步骤可以看出，日志切换包含了很多工作，而且在整个过程中，日志的生成都是被完全禁止的，因此在这期间，对数据库的修改操作会被全部阻塞。因此我们常说："日志切换是一种较为昂贵的操作。"既然日志切换十分昂贵，对系统性能的影响较大，那么就应该想办法减少日志切换的数量，提高日志切换的速度。

减少日志切换的数量可以从两个方面入手，一方面是减少日志的产生量，另一方面是加大日志文件的大小。

对于减少日志产生量，常规的办法不外乎使用 nologging 操作，如 BULK 操作、direct path write 操作等。不过大家要注意，在归档模式和非归档模式下，这些 nologging 操作的效果是不同的。Julian Dyke 对常见的可以使用 nologging 模式的操作进行过测试，他在一个归档模式的数据库中，对一张 10 万条记录的表进行了一系列的操作，测试结果如表 5-1 所示。

表　5-1

Operation	LOGGING	NOLOGGING
CREATE TABLE AS SELECT	14238844	39548
ALTER TABLE MOVE	14227236	45340
INSERT /*+ APPEND */	14221904	42452
CREATE MATERIALIZED VIEW	20726784	3784532
CREATE INDEX	2042532	24548
ALTER INDEX REBUILD	2056440	32192
ALTER INDEX REBUILD ONLINE	2083832	67840
SQL*Loader(Direct)	14248116	56712
Online Reorganization	21330788	7169472

可以看出，nologging 操作的效果还是十分明显的。CTAS 操作的 REDO 日志量从 14MB 下降到 39KB 左右。老白曾经也做过一个类似的测试，测试环境是在一个非归档模式的数据库中，通过 DBA_OBJECTS 生成了一张具有 10 万条记录的表。做 CTAS 操作时，在没有使用 nologging

的情况下，产生的 REDO 量是 113352B，在使用 nologging 的情况下，产生的 REDO 量是 176840B。从数据上看，nologgging 并没有减少 REDO 产生量，在非归档模式下，像 CTAS 这样的操作本身产生的 REDO 量就很有限，因此 nologging 的作用就不大了。

另外，在归档模式下，direct path write 操作的 REDO 量大幅减少了，这是什么原因导致的呢？我们通过转储 REDO 日志来看一看 INSERT /*+ APPEND */操作产生的 REDO 和普通的 INSERT SELECT 操作产生的 REDO 有什么不同。

```
REDO RECORD - Thread:1 RBA: 0x0006d1.0000003f.0130 LEN: 0x2050 VLD: 0x01
SCN: 0x0000.053653c2 SUBSCN:  2 09/21/2010 09:56:40
CHANGE #1 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:24.6
CHANGE #2 TYP:1 CLS: 1 AFN:10 DBA:0x0280c406 OBJ:122858 SCN:0x0000.053653bd SEQ:  1
    OP:19.1
Direct Loader block redo entry
Block header dump:  0x00000000
 Object id on Block? Y
 seg/obj: 0x1dfea  csc: 0x00.53653bc  itc: 3  flg: E  typ: 1 - DATA
     brn: 0  bdba: 0x280c38a ver: 0x01 opc: 0
     inc: 0  exflg: 0

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0001.00a.000054d6  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
0x03   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000

data_block_dump,data header at 0xb68ab288
===============
tsiz: 0x1f80
hsiz: 0xc8
pbl: 0xb68ab288
bdba: 0x00000000
     76543210
flag=--------
ntab=1
nrow=91
frre=-1
fsbo=0xc8
fseo=0x432
avsp=0x36a
tosp=0x36a
0xe:pti[0]      nrow=91 offs=0
0x12:pri[0]     offs=0x1f36
0x14:pri[1]     offs=0x1eea
0x16:pri[2]     offs=0x1ea1
0x18:pri[3]     offs=0x1e57
0x1a:pri[4]     offs=0x1e09
0x1c:pri[5]     offs=0x1dbe
0x1e:pri[6]     offs=0x1d69
0x20:pri[7]     offs=0x1d1e
0x22:pri[8]     offs=0x1cd2
0x24:pri[9]     offs=0x1c79
0x26:pri[10]    offs=0x1c2f
......
```

可以看到，在归档模式下，direct path load 操作将整个数据块放入 REDO 日志中，这样既大幅减少了 REDO 的产生量，又保证了不会丢失数据。在非归档模式下，direct path load 操作产生的日志为：

```
REDO RECORD - Thread:1 RBA: 0x0006d0.0000bf37.00fc LEN: 0x0060 VLD: 0x01
SCN: 0x0000.05364c72 SUBSCN:  2 09/21/2010 09:45:57
CHANGE #1 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:24.6
CHANGE #2 INVLD AFN:10 DBA:0x0280c386 BLKS:0x0003 SCN:0x0000.05364c72 SEQ:  1 OP:19.2
Direct Loader invalidate block range redo entry
```

在这种情况下，如果数据库产生故障，这个数据块就无法恢复了，可能成为坏块。

除了减少 REDO 日志产生量以外，减少日志切换还可以从加大 REDO 日志文件的大小入手。比如，一个每秒产生 1MB 数据的系统，每分钟产生的数据是 60MB，如果 REDO 日志文件的大小是 120MB，那么平均 2 分钟就会产生一次日志切换。而如果我们把 REDO 日志文件的大小增加到 600MB，那么平均 10 分钟才会产生一次日志切换。

有些 DBA 担心加大 REDO 日志文件后会增加数据丢失的风险。的确，REDO 日志文件越大，其所包含的 REDO 记录的数量就越多。一旦整个 REDO 日志文件丢失或者损坏，可能丢失的数据量就会增加。实际上，整个 REDO 日志丢失的可能性极小，最可能发生的是 REDO 日志文件被误删。而如果存储出现故障，导致 REDO 日志文件损坏，那么受影响的肯定将是所有的 REDO 日志文件，而不是某一个文件。此时无论 REDO 日志信息是存储在一个 REDO 日志文件中，还是存储在两个 REDO 日志文件中，其结果都是完全一样的。

还有一种最常见的情况，就是服务器突然宕机，这时 REDO 日志文件的大小不同可能造成的数据丢失是否也会不同呢？首先我们要了解一下服务器宕机时可能丢失的数据有哪些。如果是已经提交的数据，CHECKPOINT 已经推进到的部分，就已经被写入数据文件了，这部分数据是无论如何都不会丢失的，REDO 日志是用来恢复最后一次 CHECKPOINT 到宕机前被写入 REDO 日志文件的那部分数据。由于 REDO 日志文件的写入是顺序的，所以无论这部分数据被写入到一个文件还是多个文件，都不影响数据的恢复。因此可以看出，REDO 日志文件的大小和服务器宕机丢失数据的数量是无关的。

通过前面的分析我们应该已经了解到，在绝大多数情况下，加大 REDO 日志文件的大小并不会增加数据丢失的风险。因此我们在考虑 REDO 日志文件大小的时候，基本上可以忽略数据丢失问题。

不过在某些情况下，在需要加大 REDO 日志文件时，需要注意下列问题。一是在有数据保护的情况下，为了减少故障切换时的数据丢失量，不宜将 REDO 日志文件设置得过大。另外，如果存在 CDC 或者流复制下游捕获的环境，就需要考虑 REDO 日志文件大小和捕获延时的关系问题。

很多 DBA 都认为，应该尽可能地保证 REDO 日志切换的时间不低于 10 分钟或 20 分钟。如果日志切换间隔太低，就要考虑加大 REDO 日志文件的大小。事实上，这里并没有任何铁律，只要日志切换没有对系统的性能和安全产生严重的影响，那么哪怕 1 分钟切换 2 次日志又有什么关系呢？

5.2.5　事务提交和回滚的过程

以前有人在面试 DBA 时经常会问，1 万行记录的提交和 1 行记录的提交在速度上是否相同。要想回答这个问题，就需要了解 Oracle 提交的机制，本节老白就和大家一起来讨论这种机制。

提交（COMMIT）操作的内部处理过程其实很简单。Oracle 服务进程使用一种快速提交机制来确保提交的变化已经被 Oracle 数据库接受，这样，即使在实例故障时也能够通过内部机制来恢复。Oracle 服务进程在执行一个事务时会不断地修改数据，将产生的 REDO 信息写入日志缓冲区，而 LOG WRITER 进程（LGWR）则负责不断地将日志缓冲区中的数据写入 REDO 日志文件。不管这个事务修改的数据块是否已经存盘，只要 REDO 日志信息存盘了，那么我们就认为这个数据也已经完成了存盘。因为即便此时数据库实例发生故障，下一次数据库实例启动时也可以根据 REDO 日志文件里的内容进行恢复。当事务提交的时候，Oracle 服务器会为这个事务分配一个 SCN。根据这个 SCN，服务进程会产生一个提交记录，该记录包含了 SCN。然后提交记录被写入 REDO 日志缓冲区，此时服务进程就需要 LGWR 进程“帮忙”了。它会通知并激活 LGWR 进程，后者将日志缓冲区中的数据写入 REDO 日志文件。在 LGWR 进程完成写入之前，服务进程会等待 LOG FILE SYNC 事件。在 LGWR 进程将未写入 REDO 日志文件的该提交记录之前（包含该记录）的所有 REDO 日志缓冲区都写入 REDO 日志文件后，服务进程就认为提交已经完成了，它会通过 TWO TASK 通信机制通知客户端“本事务已经提交完成，这些数据不会丢失了”。通知信息发出后，实际上本次事务并没有彻底完成，此时服务进程还有很多的工作要做，比如释放锁资源、清理 ITL 等。不过这些工作都可以稍后再进行，客户端已经收到了事务提交完成的信息，可以继续其他的工作了。

图 5-1 就是事务提交过程的示意图，这个图来自 OCP 培训讲义，这里老白偷个懒，直接引用如下。

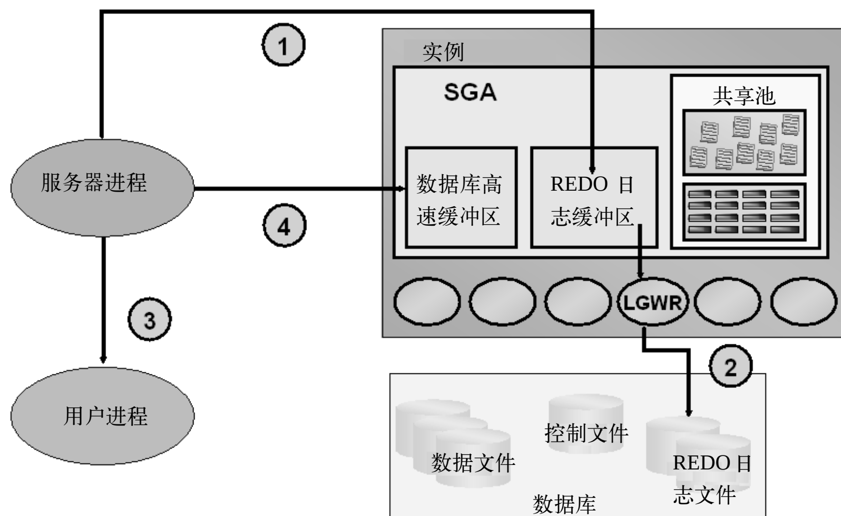


图　5-1

图 5-1 中，第一步是服务器进程将提交记录写入 REDO 日志缓冲区，第二步是 LGWR 进程将相关的 REDO 记录写入 REDO 日志文件，第三步是服务器进程将提交完成的消息通知用户进程，第四步是服务器进程继续进行后续处理。从上面的图中我们可以看出，决定提交操作速度的因素实际上是前面三个步骤，其中第一步和第三步基本相同，唯一的区别是第二步 LGWR 进程将日志缓冲区写入 REDO 日志文件的等待时间。如果 REDO 日志缓冲区中的数据积压不多，那么这个时间几乎可以忽略不计，一般在 1 ~ 4ms。如果 REDO 日志缓冲区中积压了较多的数据，那么 LOG FILE SYNC 等待的时间就会略长一些，不过在系统 I/O 性能较为正常的时候，这个等待一般在 10ms 以内。因此我们可以说，无论事务大小，其提交的速度基本上相差无几。

反过来，对于回滚（ROLLBACK）操作，就没这么简单了，因为它需要将已经修改的数据恢复到修改前的状态，因此回滚操作会因为事务的大小不同而产生很大的差异。一个修改了数百万记录的事务，回滚操作可能需要数分钟，甚至几十分钟。

5.3　REDO 优化

我们在学习 REDO 日志的基本原理和算法时，一个很重要的目的就是为了进行相关的优化工作。REDO 的优化实际上要从两个角度去考虑，一个是如何让应用尽可能减少 REDO 的产生量，另外一个是如何加快 REDO 日志存盘的速度。首先，我们来分析如何减少 REDO 日志的产生量。

5.3.1　BULK 操作能减少 REDO 吗

记得 10 多年前，老白为电信开发计费账务系统时，电信的计费系统应该算是“海量”数据处理系统了。一个本地网可能拥有 50 万电话用户，其中 20 ~ 30 万用户是有长话业务权限的，这些用户每个月可能会产生 500 万以上的话单。每个月底如何处理这些话单就是一个很大的挑战。1999 年，电信总局发布了电信账务系统业务规范，并且要求各个开发商将其开发的账务系统统一拿到电信总局去测试，测试通过的才能够获得入网许可。在参加测试的 60 多家企业中，老白设计的系统虽然用户界面比较丑陋（当时老白的公司一共也就七八条“枪”，整个开发团队只有 7 个人，因此 UI 方面和其他动辄几十人开发团队的公司是没法比的），不过在整个账务处理的性能上是首屈一指的。在包含 50 万长话用户、500 万话单记录的账务处理中，总耗时只有 4 小时多，拿到了第一名，而第二名的成绩是 6 个半小时，第三名的成绩就已经是 10 小时开外了。当时老白能够胜出的法宝有两个：一个是将 50 万用户资料一次性载入内存，在内存中通过 B 树结构保存；第二个就是使用了 BULK 操作。后来老白和第二名的公司进行了沟通，得知他们相似的地方是也做了 50 万用户资料的预装载，但是没有使用 BULK 操作。

BULK 插入操作比普通插入操作的速度要快很多，这一点是很多使用过 BULK 操作的人都了解的，不过为什么 BULK 操作会比较快呢？Oracle 官方的说法是，进行 BULK 操作的时候，USER 进程和 SQL 引擎的交互次数会大大减少，因此 BULK 操作有较好的性能。老白也一直认同这个观点，不过有一点疑惑的是，BULK 操作和普通操作的差异仅仅在于和 SQL 引擎的交互次数上吗？难道 BULK 操作是一次性向 SQL 引擎提交一个 SQL，SQL 引擎内部处理 BULK 操作时是将

整个数组还原为若干条记录去插入的吗？还是说 BULK 插入在 Oracle 内部处理过程中有一些独特的地方呢？

在研究 REDO OPCODE 的时候，老白发现了一些蛛丝马迹。LAYER 11 是针对 ROW ACCESS 方面的，也就是处理行数据的。在 LAYER 11 中，有这样一些操作，如代码清单 5-2 所示。

代码清单 5-2

```
Layer 11 : Row Access -  KCOCODRW     [kdocts.h]
         Opcode 1 : Interpret Undo Record (Undo)
         Opcode 2 : Insert Row Piece
         Opcode 3 : Drop Row Piece
         Opcode 4 : Lock Row Piece
         Opcode 5 : Update Row Piece
         Opcode 6 : Overwrite Row Piece
         Opcode 7 : Manipulate First Column (add or delete the 1rst column)
         Opcode 8 : Change Forwarding address
         Opcode 9 : Change the Cluster Key Index
         Opcode 10 : Set Key Links (change the forward & backward key links
                     on a cluster key)
         Opcode 11 : Quick Multi-Insert
         Opcode 12 : Quick Multi-Delete
         Opcode 13 : Toggle Block Header flags
```

我们注意到，11.11 的定义为 Quick Multi-insert，11.12 为 Quick Multi-Delete，这两个 OPCODE 是不是有可能和 BULK 操作有关呢？我们来做一个实验，首先创建一张测试表，如代码清单 5-3 所示。

代码清单 5-3

```
drop table sm_histable0101;
CREATE TABLE SM_HISTABLE0101
(
  SM_ID              NUMBER(10)                 NOT NULL,
  SM_SUBID           NUMBER(3)                  NOT NULL,
  SERVICE_TYPE       VARCHAR2(6),
  ORGTON             NUMBER(3),
  ORGNPI             NUMBER(3),
  ORGADDR            VARCHAR2(21)               NOT NULL,
  DESTTON            NUMBER(3),
  DESTNPI            NUMBER(3),
  DESTADDR           VARCHAR2(21)               NOT NULL,
  PRI                NUMBER(3),
  PID                NUMBER(3),
  SRR                NUMBER(3),
  DCS                NUMBER(3),
  SCHEDULE           VARCHAR2(21),
  EXPIRE             VARCHAR2(21),
  FINAL              VARCHAR2(21),
  SM_STATUS          NUMBER(3),
  ERROR_CODE         NUMBER(3),
  UDL                NUMBER(3),
```

```
  SM_TYPE            NUMBER(10),
  SCADDRTYPE         NUMBER(3),
  SCADDR             VARCHAR2(21),
  MOMSCADDRTYPE      NUMBER(3),
  MOMSCADDR          VARCHAR2(21),
  MTMSCADDRTYPE      NUMBER(3),
  MTMSCADDR          VARCHAR2(21),
  SCHEDULEMODE       NUMBER(3),
  UD                 VARCHAR2(255),
  ID_HINT            NUMBER(10)                  NOT NULL,
  DELIVERCOUNT       NUMBER(10),
  L2CACHE            NUMBER(10),
  L2CACHEWRITECOUNT  NUMBER(10),
  SERVICE            NUMBER(10),
  NEWORGADDRESS      VARCHAR2(21),
  NEWDESTADDRESS     VARCHAR2(21)
);
```

然后创建两个存储过程 REDO1 和 REDO2，分别用于普通插入操作和 BULK 插入操作，如代码清单 5-4 和代码清单 5-5 所示。

代码清单 5-4

```
create or replace procedure redo1 is
  TYPE T_SM_ID    IS TABLE OF         NUMBER(10)   INDEX BY BINARY_INTEGER;
  TYPE T_SM_SUBID IS TABLE OF         NUMBER(3)    INDEX BY BINARY_INTEGER;
  TYPE T_ORGADDR    IS TABLE OF       VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_DESTADDR   IS TABLE OF       VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_ID_HINT    IS TABLE OF       NUMBER(10)   INDEX BY BINARY_INTEGER;
  V_SM_ID         T_SM_ID;
  V_SM_SUBID      T_SM_SUBID;
  V_ORGADDR       T_ORGADDR;
  V_DESTADDR      T_DESTADDR;
  V_ID_HINT       T_ID_HINT;
  I INTEGER;
  VREDO1 INTEGER;
  vredo2 integer;
BEGIN
   FOR I IN 1.. 2000
   LOOP
      V_SM_ID(I):=I;
      V_SM_SUBID(I):=12;
      V_ORGADDR(I):='444555565';
      V_DESTADDR(I):='555555';
      V_ID_HINT(I):=i;
   END LOOP;
   select value  into vredo1 from v$sysstat where name = 'redo size';
   FOR I IN 1..2000 LOOP
     INSERT INTO SM_HISTABLE0101 (SM_ID,SM_SUBID,ORGADDR,DESTADDR,ID_HINT) VALUES
       (V_SM_ID(I),V_SM_SUBID(I),V_ORGADDR(I),V_DESTADDR(I),V_ID_HINT(I));
   END LOOP;
   COMMIT;
   commit;
   select value  into vredo2 from v$sysstat where name = 'redo size';
```

```
      select value  into vredo2 from v$sysstat where name = 'redo size';
      dbms_output.put_line('redo size:'||to_char(vredo2-vredo1));

   END;
   /
```

代码清单 5-5

```
create or replace procedure redo2 is
  TYPE T_SM_ID    IS TABLE OF          NUMBER(10)   INDEX BY BINARY_INTEGER;
  TYPE T_SM_SUBID IS TABLE OF          NUMBER(3)    INDEX BY BINARY_INTEGER;
  TYPE T_ORGADDR    IS TABLE OF        VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_DESTADDR   IS TABLE OF        VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_ID_HINT    IS TABLE OF        NUMBER(10)   INDEX BY BINARY_INTEGER;
  V_SM_ID         T_SM_ID;
  V_SM_SUBID      T_SM_SUBID;
  V_ORGADDR       T_ORGADDR;
  V_DESTADDR      T_DESTADDR;
  V_ID_HINT       T_ID_HINT;
  I INTEGER;
  VREDO1 INTEGER;
  vredo2 integer;
  n integer;
BEGIN
   n:=2000;
   FOR I IN 1.. N
   LOOP
      V_SM_ID(I):=I;
      V_SM_SUBID(I):=12;
      V_ORGADDR(I):='444555565';
      V_DESTADDR(I):='555555';
     V_ID_HINT(I):=i;
   END LOOP;
   select value  into vredo1 from v$sysstat where name = 'redo size';

   FORALL I IN 1..N
     INSERT INTO SM_HISTABLE0101 (SM_ID,SM_SUBID,ORGADDR,DESTADDR,ID_HINT) VALUES
       (V_SM_ID(I),V_SM_SUBID(I),V_ORGADDR(I),V_DESTADDR(I),V_ID_HINT(I));
   COMMIT;
   commit;
   select value  into vredo2 from v$sysstat where name = 'redo size';
   select value  into vredo2 from v$sysstat where name = 'redo size';
   dbms_output.put_line('redo size:'||to_char(vredo2-vredo1));
END;
/
```

然后执行代码清单 5-6 所示的测试。

代码清单 5-6

```
set serveroutput on
truncate table sm_histable0101;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;
exec redo1;
```

```
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;


truncate table sm_histable0101;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;
exec redo2;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;
```

测试结果如下：

```
Table truncated.

SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87596111

SQL> redo size:707356

PL/SQL procedure successfully completed.

SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87596151

SQL> truncate table sm_histable0101;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;
exec redo2;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;

Table truncated.

SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87596178

SQL> redo size:138728

PL/SQL procedure successfully completed.

SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87596195
```

从测试的结果来看，使用普通插入操作产生了 707356B 的 REDO 日志，而使用 BULK 插入操作只产生了 138728B 的 REDO 日志，REDO 日志产生量不到正常水平的 1/5。看样子 BULK 插入在 Oracle RDBMS 内部的操作是完全不同的，应该是采用了前面所猜测的 QUICK MULTI-INSERT 操作。我们可以通过转储 REDO 日志来验证一下：

```
SQL> alter system dump logfile '/opt/oracle/oradata/orcl/redo01.log' scn min 87596178
    scn max 87596195;

System altered.
```

转储出的 REDO 信息如下：

```
CHANGE #2 TYP:0 CLS:18 AFN:7 DBA:0x01c007f2 OBJ:4294967295 SCN:0x0000.05389bf0
    SEQ:  2 OP:5.1
ktudb redo: siz: 396 spc: 5858 flg: 0x0012 seq: 0x2aba rec: 0x11
            xid:  0x0001.021.0000551d
ktubl redo: slt: 33 rci: 0 opc: 11.1 objn: 122951 objd: 122956 tsn: 0
Undo type:  Regular undo        Begin trans    Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
             0x00000000  prev ctl uba: 0x01c007f2.2aba.0f
prev ctl max cmt scn:  0x0000.05388a1f  prev tx cmt scn:  0x0000.05388a26
txn start scn:  0xffff.ffffffff  logon user: 0  prev brb: 29362160  prev bcl: 0
    KDO undo record:
KTB Redo
op: 0x03  ver: 0x01
op: Z
KDO Op code: QMD row dependencies Disabled
  xtype: XA flags: 0x00000000  bdba: 0x00406482  hdba: 0x00406481
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 lock: 0 nrow: 131
slot[0]: 0
slot[1]: 1
slot[2]: 2
slot[3]: 3
slot[4]: 4
slot[5]: 5
slot[6]: 6
slot[7]: 7
slot[8]: 8
slot[9]: 9
slot[10]: 10
slot[11]: 11
slot[12]: 12
slot[13]: 13
slot[14]: 14
slot[15]: 15
slot[16]: 16
slot[17]: 17
slot[18]: 18
slot[19]: 19
slot[20]: 20
slot[21]: 21
slot[22]: 22
slot[23]: 23
slot[24]: 24
```

从上述信息来看，UNDO 的数据也和普通的插入操作不同，是批量方式的，同一个数据块中的所有记录都产生在同一个 UNDO 的 CHANGE VECTOR 中。接下来，我们再来分析代码清单 5-7 所示的数据。

代码清单 5-7

```
CHANGE #3 TYP:0 CLS: 1 AFN:1 DBA:0x00406482 OBJ:122956 SCN:0x0000.05389c94 SEQ:  3
OP:11.11
KTB Redo
op: 0x01  ver: 0x01
op: F  xid:  0x0001.021.0000551d    uba: 0x01c007f2.2aba.11
KDO Op code: QMI row dependencies Disabled
  xtype: XA flags: 0x00000000  bdba: 0x00406482  hdba: 0x00406481
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 lock: 1 nrow: 131
slot[0]: 0
tl: 53 fb: --H-FL-- lb: 0x0  cc: 29
col  0: [ 2]  c1 02
col  1: [ 2]  c1 0d
col  2: *NULL*
col  3: *NULL*
col  4: *NULL*
col  5: [ 9]  34 34 34 35 35 35 35 36 35
col  6: *NULL*
col  7: *NULL*
col  8: [ 6]  35 35 35 35 35 35
col  9: *NULL*
col 10: *NULL*
col 11: *NULL*
col 12: *NULL*
col 13: *NULL*
col 14: *NULL*
col 15: *NULL*
col 16: *NULL*
col 17: *NULL*
col 18: *NULL*
col 19: *NULL*
col 20: *NULL*
col 21: *NULL*
col 22: *NULL*
col 23: *NULL*
col 24: *NULL*
col 25: *NULL*
col 26: *NULL*
col 27: *NULL*
col 28: [ 2]  c1 02
slot[1]: 1
tl: 53 fb: --H-FL-- lb: 0x0  cc: 29
```

可以看出，这里确实使用了 OP CODE:11.11，BULK 插入操作使用了批量数据插入机制，因此其性能才能够远高于单条记录操作。在我们以往进行的测试中，BULK 操作一般都比单条操作快 1 倍以上，有时甚至能够达到 2 倍以上。

在 REDO LAYER 11 中，大家也许会发现一个问题，11.11 是 MULTI-INSERT，11.12 是 MULTI-DELETE，唯独缺少了 MULTI-UPDATE，难道 BULK 更新操作的实现机制和 BULK 插入操作有所不同吗？我们通过一个实验来看看 BULK 更新是否能够减少 REDO 的产生量。通过简

单地修改 REDO1、REDO2 两个存储过程，生成 REDOU1、REDOU2 这两个存储过程：

```
create or replace procedure redou1 is
  TYPE T_SM_ID    IS TABLE OF          NUMBER(10)   INDEX BY BINARY_INTEGER;
  TYPE T_SM_SUBID IS TABLE OF          NUMBER(3)    INDEX BY BINARY_INTEGER;
  TYPE T_ORGADDR    IS TABLE OF        VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_DESTADDR   IS TABLE OF        VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_ID_HINT    IS TABLE OF        NUMBER(10)   INDEX BY BINARY_INTEGER;
  V_SM_ID         T_SM_ID;
  V_SM_SUBID      T_SM_SUBID;
  V_ORGADDR       T_ORGADDR;
  V_DESTADDR      T_DESTADDR;
  V_ID_HINT       T_ID_HINT;
  I INTEGER;
  VREDO1 INTEGER;
  vredo2 integer;
BEGIN
   FOR I IN 1.. 2000
   LOOP
      V_SM_ID(I):=I;
      V_SM_SUBID(I):=12;
      V_ORGADDR(I):='111111';
      V_DESTADDR(I):='2222';
      V_ID_HINT(I):=i;
   END LOOP;
   select value  into vredo1 from v$sysstat where name = 'redo size';
   FOR I IN 1..2000 LOOP
     update SM_HISTABLE0101 SET ORGADDR=V_ORGADDR(I) WHERE ID_HINT=V_ID_HINT(I);
   END LOOP;
   COMMIT;
   commit;
   select value  into vredo2 from v$sysstat where name = 'redo size';
   select value  into vredo2 from v$sysstat where name = 'redo size';
   dbms_output.put_line('redo size:'||to_char(vredo2-vredo1));

END;
/

create or replace procedure redoU2 is
  TYPE T_SM_ID    IS TABLE OF          NUMBER(10)   INDEX BY BINARY_INTEGER;
  TYPE T_SM_SUBID IS TABLE OF          NUMBER(3)    INDEX BY BINARY_INTEGER;
  TYPE T_ORGADDR    IS TABLE OF        VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_DESTADDR   IS TABLE OF        VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_ID_HINT    IS TABLE OF        NUMBER(10)   INDEX BY BINARY_INTEGER;
  V_SM_ID         T_SM_ID;
  V_SM_SUBID      T_SM_SUBID;
  V_ORGADDR       T_ORGADDR;
  V_DESTADDR      T_DESTADDR;
  V_ID_HINT       T_ID_HINT;
  I INTEGER;
  VREDO1 INTEGER;
  vredo2 integer;
  n integer;
BEGIN
```

```
   n:=2000;
   FOR I IN 1.. N
   LOOP
      V_SM_ID(I):=I;
      V_SM_SUBID(I):=12;
      V_ORGADDR(I):='111111';
      V_DESTADDR(I):='2222';
     V_ID_HINT(I):=i;
   END LOOP;
   select value  into vredo1 from v$sysstat where name = 'redo size';

   FORALL I IN 1..N
     update SM_HISTABLE0101 SET ORGADDR=V_ORGADDR(I) WHERE ID_HINT=V_ID_HINT(I);
   COMMIT;
   commit;
   select value  into vredo2 from v$sysstat where name = 'redo size';
   select value  into vredo2 from v$sysstat where name = 'redo size';
   dbms_output.put_line('redo size:'||to_char(vredo2-vredo1));
END;
/
```

然后执行下面的过程：

```
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;
exec redou1;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;

select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;
exec redou2;
select max(ktuxescnw * power(2, 32) + ktuxescnb)  from x$ktuxe;

redo size:578904

PL/SQL procedure successfully completed.

SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87608317

SQL> SQL> SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87608317

SQL> redo size:571168

PL/SQL procedure successfully completed.

SQL>
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
------------------------------------
                            87610350
```

从上面的结果来看，两种操作产生的 REDO 量是十分接近的，因此 BULK 更新操作在 REDO 方面并没有很大的改善。通过转储 REDO 日志，我们进一步验证这个测试结果：

```
CHANGE #3 TYP:2 CLS: 1 AFN:1 DBA:0x00406482 OBJ:122959 SCN:0x0000.0538cbfd SEQ:  1
OP:11.5
KTB Redo
op: 0x11  ver: 0x01
op: F  xid:  0x0003.011.0000724a    uba: 0x00800ac1.32bc.1e
Block cleanout record, scn:  0x0000.0538cbff ver: 0x01 opt: 0x02, entries follow...
  itli: 2  flg: 2  scn: 0x0000.0538cbfd
KDO Op code: URP row dependencies Disabled
  xtype: XA flags: 0x00000000  bdba: 0x00406482  hdba: 0x00406481
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 slot: 0(0x0) flag: 0x2c lock: 1 ckix: 191
ncol: 29 nnew: 1 size: 0
col  5: [ 6]  31 31 31 31 31 31
```

我们在这里看到了 OP CODE 11.5，这是一个正常的单行更新操作。也就是说，对于 REDO 产生而言，FORALL 更新并不会批量产生 REDO 信息，和普通的更新处理没有区别。FORALL 更新在性能上的提升可能仅限于 SQL 引擎之间的交互而已。

5.3.2　如何优化 LOG FILE SYNC 等待事件

在一个提交十分频繁的系统中，我们经常会看到 LOG FILE SYNC 等待事件出现在 TOP EVENTS 中。这种情况下，可能就需要针对 LOG FILE SYNC 等待事件进行优化了。

首先我们会看一下这个等待事件的平均等待时长。正常情况下平均等待时间不会超过 10ms，如果等待时间太长，那说明 LOG WRITER 每次写入的时间过长，如果能够优化 REDO 日志文件的存储，使之存放到更快的磁盘上，就可以减少这个等待事件的单次等待时间。不过理论上能够实现的事情往往是最不可靠的，系统管理员可能会和你说，目前的存储优化已经做得好到不能再好了，想要存放到更快磁盘上的可能性几乎不存在。也许你很幸运，当前的 REDO 日志是存放在 RAID 5 上的，而存储上正好有个磁盘组是 RAID 10 的，而且可以划些空间给你，这样就可以通过提升 REDO 日志写性能来优化这个等待事件了。

但如果不凑巧，我们无法通过优化 REDO 日志的 I/O 性能来解决这个问题，或者优化了 REDO 日志的 I/O 性能后还是无法达到我们的预期，那么该如何处理呢？有经验的 DBA 可能会建议加大日志缓冲区。提到加大日志缓冲区，可能有些朋友就会感到疑惑了，REDO 日志文件写等待时间长怎么会和日志缓冲区直接关联起来呢？实际上这个问题解释起来一点也不难，如果数据文件的 I/O 性能有问题，平均单块读的等待时间偏长，那么可以通过加大 DB Cache 来减少 I/O 总次数，从而达到优化 I/O 的效果。加大日志缓冲区的原理也是一样的，这样可以使日志缓冲区中存储更多的 REDO 日志数据，从而减少由于 REDO 日志缓冲区不足而产生的 LGWR 写操作的数量，使平均每次写入 REDO 日志文件的 REDO 字节数增加，从而减少 REDO 的 I/O 次数，进而达到优化 LOG FILE SYNC 等待事件的目的。

如果上述两种方法都不奏效，又该如何处理呢？还有一种方法，就是减少提交的次数。如果

提交过于频繁，那么无论怎样优化都无法彻底解决问题。通过加大一次提交记录的数量，减少提交批次，可以有效地减少 LOG FILE SYNC 等待时间。采用此方法就意味着需要对应用进行较大的调整，甚至要对应用架构做出修改，这种修改的代价将十分巨大。

还有一个方案可以优化 LOG FILE SYNC 事件，就是把部分经常提交的事务设置为异步提交。异步提交是 10*g* 版本引入的新特性，通过设置 COMMIT_WRITE 参数，可以控制异步提交。COMMIT_WRITE 参数的默认值是“IMMEDIATE,WAIT”，可以将其设置为“IMMEDIATE,NOWAIT”来实现异步提交。这个参数支持系统级设置，也支持会话级设置。

如果觉得设置 COMMIT_WRITE 参数太麻烦，也可以直接使用 COMMIT WRITE 命令，比如，COMMIT WRITE IMMEDIATE NOWAIT。COMMIT WRITE 的语法如图 5-2 所示。

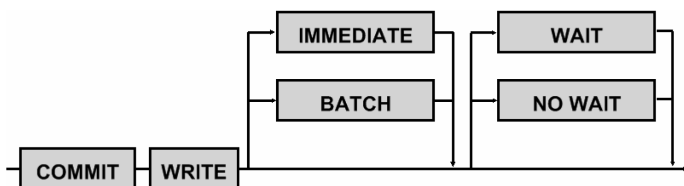


图　5-2

其中，IMMEDIATE 表示提交时 LGWR 马上开始 REDO 日志写操作，而 BATCH 表示提交后 LGWR 不需要马上开始写 REDO 日志，可以按照其原有的计划启动写操作。BATCH 可以让 REDO 日志写操作更加滞后，一般我们还是希望尽快写入 REDO 日志数据，因此“IMMEDIATE,NOWAIT”是较为常用的优化方案。

不过使用异步提交会带来一个问题。从 REDO 日志的基本原理来看，数据一旦提交成功，客户端就可以认为其已经被正常存入数据库了，哪怕实例出了故障，这些数据也不会丢失。而异步提交的数据，即使已经提交成功，也还是有可能丢失。因此一旦数据库实例有故障，采用异步提交的系统需要做一些额外的校验和处理，清理不一致的数据，重新插入刚才由于异步提交而丢失的数据。这就需要应用对当前最后一笔提交的数据进行特殊处理，建立校验机制和错误数据处理机制。我们需要在应用层面进行一些特殊的设置。应该注意的是，那些特别重要的、后续无法重新完全补充的数据不适合使用这种方法。

老白曾经参加过一个项目的投标工作，客户在进行压力测试，而测试的存储系统性能不佳，REDO 日志存放在 RAID 5 上，一旦系统负载增加到 800TPS，LOG FILE SYNC 的等待就十分严重，这也成为系统的一个主要瓶颈。由于仅仅是压力测试，实际生产系统不可能达到 800tps，而且实际中的存储系统要远好于当前的测试平台。但由于目前这个问题的存在，客户的很多测试项目无法完成。为了完成本次测试，老白就建议客户使用异步提交。采用异步提交后，系统并发处理能力明显提高，最终完成了 1200tps 的测试工作。

最后，老白要说的是，LOG FILE SYNC 等待事件是十分关键的，我们在数据库的日常维护中应该对此指标建立基线，如果这个指标有异常变化，一定要尽快分析并解决问题。一旦这个指标恶化，可能导致系统性能急剧下降，甚至会导致短暂的挂起。去年，一个客户的系统，平时

LOG FILE SYNC 的指标是 2 ~ 3ms。在一次巡检时老白发现该指标增长到了 7ms，当时在巡检报告中建议客户关注这个指标，并尽快检查存储系统和操作系统，查出变慢的原因。客户检查了存储，没有发现故障，于是就不了了之了。在下个月巡检的时候，老白发现该指标增长到了 13ms，再次预警，依然没有发现问题。随后两个月这个指标一直持续恶化，增长到了 20 多毫秒。由于前面几个月的检查工作没有发现问题，而目前系统也还是很正常的，所以客户也就没有再去认真核查。终于有一天，系统突然挂起了，5 分钟后才恢复正常。后来检查原因，就是 LOG FILE SYNC 等待导致的。根据老白的建议，客户从头到尾检查了一遍，最终发现 LVM 的一条链路存在闪断现象，修复了链路后，一切都恢复正常了。

通过上面的案例，我们要吸取教训，如果 LOG FILE SYNC 指标有所恶化，一定要尽快排查问题的根源，如果 LOG FILE SYNC 的等待时间持续上升，那么系统出现挂起的可能性也在增加。尽快找到问题的原因是势在必行的。

5.3.3　SHUTDOWN ABORT 无害吗

以前老前辈一直教育我们，千万不要用 SHUTDOWN ABORT 命令关闭数据库，因为这样会使数据库宕机。这个观点一直都影响着我，我也一直没有怀疑过“SHUTDOWN ABORT 有害”这一说法，而且在给其他人讲课时，也把这个观点不断地传递给一批又一批新的 DBA。

直到我认真研究了 REDO 日志和数据库恢复的原理，才明白原来 SHUTDOWN ABORT 并不那么可怕。根据实例恢复的原理，已经提交的数据肯定已被写入 REDO 日志文件了，如果实例突然宕掉，只要底层的存储没有出现故障，那么再次启动的时候，应该能够根据 REDO 日志文件进行恢复。如此说来，SHUTDOWN ABORT 应该是无害的。

确实，从原理上看，SHUTDOWN ABORT 命令的确不会导致数据库损坏，只是会丢失一些当前没有存盘的数据。当实例再次启动的时候，Oracle 可以根据 REDO 日志中的数据，重新修改并恢复丢失的数据。

虽然 Oracle 的 REDO 机制可以确保 SHUTDOWN ABORT 不会对正常的事务数据产生影响，但是我们忽略了一点，Oracle 数据库中数据文件的修改，并不都是基于事务这种机制的，还有一些数据的修改采取了特殊的方法。比如，各种位图的维护、数据文件扩展等。我们使用 SHUTDOWN ABORT 命令时，如果恰巧有这样的操作正在进行，那么下次数据库启动的时候，就可能出现一些不一致，甚至出现数据库实例无法启动的情况。另外，我们也不能忽视随处都可能遇到的 Bug。

所以，虽然从基本原理上来说，SHUTDOWN ABORT 不会导致数据库宕机，但是还是要十分谨慎地使用。作为一个具有近 20 年工作经验的 DBA，老白在这方面一直比较保守。在关闭数据库之前，尽可能做几次 CHECKPOINT（ALTER SYSTEM CHECKPOINT）。如果关闭过程很长时间都没有完成，就应该先考虑杀掉所有“LOCAL=NO”以及其他“LOCAL=YES”的前台进程，然后再观察是否能够正常关闭数据库。可以使用下面的命令杀掉所有“LOCAL=NO”的前台进程：

```
ps -ef|grep "LOCAL=NO" |awk '{print "kill -9 " $2}'|sh
```

不过还有一种可能，即便杀掉了所有的应用进程，数据库还是无法关闭。比如，我们可能看到：

```
Shutting down Instance (immediate)
    License high water mark = 12
    Thu Dec  8 18:43:16 1999
    alter database  close normal
    Thu Dec  8 18:43:17 1999
    SMON: disabling tx recovery
    SMON: disabling cache recovery
```


这些信息显示 SMON 已经结束了事务层面的回退操作，关闭了 cache 层面的恢复，也就是说 SMON 开始进行临时段回收工作了。如果系统中有大量需要回收的临时段，那么这个操作可能会持续很长时间。这时关闭数据库实例（通过 SHUTDOWN ABORT）是安全的，不会引起数据库故障。当然，用户也可以继续等待，直到 SMON 完成工作。不过 SMON 可能会由于某种原因挂起了，在等待了很长时间后还是这种状态，这时可以果断采取 SHUTDOWN ABORT 的动作。在关闭数据库的过程中，很有可能出现类似的挂起现象。

在做关闭操作之前，如果用户已经关闭了应用或者杀掉了所有的用户进程，并且完成了 CHECKPOINT，那么采用 SHUTDOWN ABORT 操作的风险就降到了最低。

5.3.4 关于 REDO 日志优化的建议

REDO 日志的优化其实并不难，随着 Oracle 新版本的发布和存储技术的发展，REDO 日志的性能问题越来越少。一般来说，我们针对 REDO 日志的优化集中在几个方面：

- REDO 日志文件的大小；
- REDO 日志组的数量；
- REDO 日志镜像；
- REDO 日志存储的性能；
- 日志缓冲区的设置。

REDO 日志切换是一种开销较大的操作，每次日志切换都会强制进行 CHECKPOINT。因此尽可能不要在业务高峰期过于频繁地进行日志切换，这样对于系统总体性能的帮助还是很大的。Oracle 建议日志文件的大小能够保证日志切换的时间在 20 分钟以上，但实际上，这在很多大型企业级应用系统中很难保证。很多企业级应用系统的 REDO 日志产生量很大，哪怕 REDO 日志文件设置为 2GB，也无法确保高峰期的日志切换时间超过 3 分钟，甚至有的连 1 分钟都不到。实际上，判断 REDO 日志文件大小是否足够的主要因素并不仅仅局限于日志切换的时间，如果过于频繁的日志切换并没有对系统性能和稳定性产生较大的影响，那么还是可以接受的。从 Oracle 10*g* 开始，Oracle 提供了一个 REDO LOG SIZE ADVISORY 的新功能，这个功能可以提供关于 REDO 日志文件大小的建议。系统将根据高峰期产生 REDO 的数量，以及系统日志切换的时间间隔，提出一个建议值，比如：

```
SQL> select optimal_logfile_size from v$Instance_recovery;

OPTIMAL_LOGFILE_SIZE
--------------------
                1954
```

这个值的单位是兆字节，也就是说，Oracle建议REDO日志文件的大小为1954MB，而当前系统的REDO日志文件大小是1GB，建议值约为当前值的两倍。这个建议可以供我们参考，由于当前系统并未由于CHECKPOINT和log file switch产生较为严重的问题，LOG FILE SYNC等待也比较正常，因此当前REDO日志文件的大小可以暂时不做调整。当然如果按照Oracle的建议加大REDO日志文件，也是可行的。

对于绝大多数系统来说，REDO日志组的数量不直接和性能产生关系。不过对于REDO切换十分频繁的数据库，比如，高峰期1分钟切换一次甚至不到1分钟切换一次的数据库，如果REDO日志组的数量偏少，将会导致所有REDO日志组耗尽，而且它们都处于活跃（active）状态（REDO日志文件中包含的变化量的SCN还大于CKTP SCN），那么日志切换就无法进行，必须等到某一组REDO LOG从活跃转为不活跃（inactive），才能继续进行日志切换。这时，整个系统的修改操作都处于挂起状态，等待日志文件切换完成（其等待事件就是LOG FILE SWITCH）。如果碰到这种情况，就必须增加REDO日志组的数量或者增加REDO日志文件的大小。

REDO日志的成员（member）数量就是REDO日志文件镜像的数量。默认情况下，Oracle的每个REDO日志组会有两个成员。这种配置下，如果某个REDO日志文件损坏，另外一个REDO日志文件可以用于恢复，不会导致数据库宕机。实际上，使用REDO日志镜像的要求十分严格，在最为严格的情况下，不同的成员不能存放于相同的卷组、磁盘组和物理硬盘，这样才能达到最佳的容错效果。不幸的是，老白碰到的所有系统都不满足这个条件，90%以上的系统REDO日志文件是在同一个卷组中分配的。一旦这个卷组出现故障，那么所有的成员也都会有故障，这完全违背了设置多个镜像的初衷。REDO日志文件会消耗大量I/O资源，因此REDO日志镜像会大幅增加系统I/O开销，也就是说，设置两个以上成员的意义并不是很大，除非系统的可用性要求特别高，且I/O能力远远高于系统负载。对于一个I/O性能存在瓶颈的系统，使用多个镜像是十分“奢侈”的，取消镜像可以节省大量的I/O资源，但会带来单点故障。在这种环境下，如何权衡好系统可用性和性能就十分重要了。

无论怎样配置，REDO日志文件都是写敏感的文件，其写I/O是频繁的小型写入操作，每次写入的数据量从几百字节到几十万字节不等，偶尔也有一些大的写入操作，可能达到几兆字节或几十兆字节。将REDO日志文件存放于性能较好的盘是十分必要的，最好将其存放在RAID 1+0的设备中，RAID 5对于高负载的系统来说可能会引起一些性能问题。如果某个系统的I/O性能存在明显的问题，特别是存在I/O热点，那么将REDO日志迁移到其他的磁盘组，从而隔离I/O，是比较简单的优化方法。

对于日志缓冲区和REDO日志I/O性能问题的关系，很多DBA可能都会觉得有些迷茫。确实，在这方面，互联网上充斥着伪科学。顾名思义，日志缓冲区是为了提高REDO日志写性能而设计的缓冲区，系统产生的REDO日志信息首先被缓冲在日志缓冲区中，等LGWR有能力处

理的时候才将缓冲区中的 REDO 日志信息写入到 REDO 日志文件中。这样就避免了大量的小型写操作，可以将小型的 REDO 日志片段组织成一个较大的写操作，统一写入 REDO 日志文件。如果某个时间段 REDO 日志信息的产生量很大，REDO 日志产生的速度大于 LGWR 写入 REDO 日志的速度，那么日志缓冲区中的数据就会积压。当积压到一定程度，日志缓冲区就可能被撑爆。这时，产生 REDO 的会话就会增加 redo buffer allocation retries 计数，这个操作将被暂时挂起，等待日志缓冲区中空闲空间的出现。因此在一个 REDO 日志生成量很大的系统中（比如每秒产生 4MB 以上，甚至 10MB 以上的系统），可能需要将日志缓冲区设置得比较大，比如 20MB、30MB 甚至上百兆字节。关于日志缓冲区超过 3MB 就没有意义的言论，老白已经在多个场合批驳过了，这里就不再浪费篇幅去讨论了。

第 6 章 理解 UNDO

UNDO 给大家的最初印象就是事务的回滚。确实，UNDO 实现了 Oracle 数据库的事务回滚。实际上，回滚事务只是 UNDO 的一个功能，UNDO 还有一个十分重要的功能就是一致性读。由于 UNDO 的存在，使得一致性读成为可能。如果没有 UNDO，那么要实现一致性读就必须将读操作定义成排他性的，当某个数据被读之前，不允许任何对该数据的修改操作，这样会大大降低并发操作的性能。由于 UNDO 的存在，一个数据的前映像（pre image）可以被存放在 UNDO 表空间中，因此在查询时，哪怕数据已经被修改了，我们仍可以从 UNDO 中找到某个时间点的前映像，从而完成一致性读。

从 Oracle 9*i* 开始，UNDO 被赋予了新的功能——闪回查询。闪回查询就是利用 UNDO 中保存的前映像查询以前的数据。除了闪回查询外，9*i* 版本还有一个新功能，就是查询某个时间段内的数据变化。这个功能实际上和闪回查询类似，都是通过前映像来实现的。

早期版本的 UNDO 管理十分复杂，DBA 也经常面临 UNDO 出现的各种问题，比如 UNDO 引起的性能问题，以及著名的 ORA-1555 错误。自从 Oracle 9*i* 引入了 UNDO 自动管理后，这些问题都变得简单了，DBA 只需设置合理的 UNDO_RETENTION 参数，然后创建充足的 UNDO 表空间，剩下的事情都可以交给 Oracle 自己去完成。很多 2000 年后入行的 DBA 都没有经历过手工管理 UNDO 的痛苦，这是十分幸福的，不过也正是因为我们很少关注 UNDO，才导致了现在很多 DBA 对 UNDO 的机制和原理一知半解，一旦碰到问题，就十分麻烦了。

在本节中，老白会带着大家简单了解 UNDO 的基本概念，并重温 UNDO 手工管理时代的一些主要优化思路。即使不需要手工管理 UNDO，了解一些这方面的知识也更有助于今后分析有关 UNDO 的问题。

6.1 UNDO 的基本原理

UNDO 是事务层面的组件，它实现了数据库的事务回滚和一致性读。从 9*i* 版本开始，UNDO 还为闪回查询提供了保证。正因为 UNDO 和应用的关系十分密切，所以我们需要深入了解 UNDO 的基本原理和算法，以便于解决维护工作中可能出现的问题。

6.1.1 UNDO 表空间和回滚段

UNDO 表空间是一个十分特殊的表空间，它不存储表和索引数据，而只存放回滚段（ROLLBACK SEGMENT）。回滚段是一种非常有趣的机制，Oracle 回滚段的作用不仅仅局限于事务的回滚，一致性读是回滚段的第二个作用。通过一致性读衍生出来的闪回查询只能算是一致性读的副产品了。

在 9*i* 版本之前，UNDO 是需要 DBA 手工管理的。在创建数据库时会默认创建一个 RBS 表空间，在 RBS 表空间中会创建几个回滚段。一般来说，DBA 还需要手工调整 RBS 的参数，同时新创建几个回滚段，以适应系统负载的需要。创建多少个回滚段，每个回滚段的初始大小是多少，最佳大小是多少，这些都需要 DBA 自己去分析和考虑。创建完回滚段后，DBA 还要手工将这些回滚段写入参数 ROLLBACK_SEGMENTS 中，否则数据库系统将不会自动使用这些回滚段。

设计回滚段对于 DBA 来说是个考验，如果回滚段配置得不合理，可能会导致性能问题，从而影响系统的稳定运行。很多 DBA 在这方面缺乏经验，因此就会导致那个十分著名的 ORA-1555 错误频发。在 8*i*、8.0 或者 7.3 版本的时代，面试官最喜欢问 DBA 的问题就是"ORA-1555 是怎么发生的，如何进行优化"。能把这个问题回答得十分完美的 DBA 水平确实很不错了。要想在手工管理模式下尽可能避免 ORA-1555 错误，需要有足够大的 RBS 空间、足够多的回滚段，每个回滚段有足够大的 OPTIMAL SIZE。即使上述条件都满足，还是很难避免 ORA-1555 的出现，因为有些查询的执行时间太长，SQL 质量之差几近变态。但现在，如果有一个面试官问前来应聘的 DBA，如何在手工管理模式下尽可能避免 ORA-1555 错误？我想能够正确回答出来的人不会很多。因为进入 9*i* 版本以后，手工管理回滚段已经变为历史。在 UNDO_MANAGEMENT=AUTO 的模式下，UNDO 管理已经相当简单了，只要确保有一个足够大的 UNDO 表空间，设置足够大的 UNDO_RETENTION，剩下的一切就可以交给 Oracle 自己去完成了。UNDO 中需要配置多少个回滚段，每个 RBS 的 OPTIMAL SIZE 多大，这些都是 RDBMS 根据并发的事务数量自动判断的。从 10*g* 版本开始，甚至连 UNDO_RETENTION 也自动管理了，Oracle 会根据 UNDO 表空间的大小和 MAX QUERY LENGTH 这些指标，自动调整 UNDO_RETENTION 的值，从而充分利用 UNDO 表空间，保留最多的前映像。因此很多 DBA 会发现，10*g* 版本的 UNDO 表空间经常出现 100% 被使用的情况，实际上这是十分正常的。在 10*g* 版本中，如果使用默认的 UNDO_RETENTION 自动管理，那么最好将 UNDO 表空间的所有文件的自动扩展属性全部关闭，否则经过一段时间的运行，你会发现 UNDO 表空间变得非常大。

回滚段用于存放数据修改之前的值（包括数据修改之前的位置和值），称为前映象。回滚段的头部包含正在使用的该回滚段事务的信息，一个事务只能使用一个回滚段来存放它的回滚信息，而一个回滚段可以存放多个事务的回滚信息。回滚段的主要用途如下。

- 事务回滚。当事务修改表中数据的时候，该数据修改前的值（即前映像）会存放在回滚段中。当用户回滚事务（ROLLBACK）时，Oracle 将会利用回滚段中的数据前映像来将已修改的数据恢复到原来的值。

- 事务恢复。当事务正在处理的时候，例程失败，回滚段的信息保存在 REDO 日志文件中，Oracle 将在下次启动数据库时利用回滚来恢复未提交的数据。
- 读一致性。当一个会话正在修改数据时，其他的会话看不到该会话未提交的修改。而且，当一个语句正在执行时，该语句将看不到开始执行后的未提交的修改（语句级读一致性）。当 Oracle 执行 SELECT 语句时，Oracle 依照当前的系统改变号（SYSTEM CHANGE NUMBER，SCN）来保证任何在当前 SCN 之前的未提交的改变不被该语句处理。不难想象，当一个长时间的查询正在执行时，若其他会话改变了该操作要查询的某个数据块，Oracle 将利用回滚段的数据前映像来构造一个读一致性视图。
- 闪回查询。从 9*i* 版本开始，Oracle 支持闪回查询，就是通过 UNDO 中的数据，查询某个 SCN 之前的数据，或者查询某两个时间点（或者 SCN）之间，某张表发生了哪些变化。

每个回滚段都包含一些扩展（EXTENTS）。回滚段采用一种循环机制来使用这些扩展，当某个扩展写满后，自动切换到另外一个扩展继续使用。一个事务会将回滚记录写在回滚段的当前位置，并且通过记录大小来标明记录的位置。当前写指针是回滚段段头中的一个控制结构。尾部（TAIL）指的是回滚段中最后一条记录的开始位置。

回滚段的数量和每个回滚段的大小对于回滚段的配置来说至关重要。在 OLTP 系统中，应设置足够的回滚段数量，以避免回滚段冲突。另外，每个回滚段的大小也应该足够大，以便于适应事务处理的需要。Oracle 循环使用回滚段，所有的在线回滚段（除了 SYSTEM 回滚段外）都会被轮流使用。Oracle 在使用回滚段的时候，遵循以下原则。

- 一个事务只使用一个回滚段来记录所有的回滚记录。
- 多个事务可以写入相同的扩展。
- 扩展可以被循环使用，且任何一个扩展都不会被跳过。
- 如果不能使用下一个扩展，Oracle 会自动分配一个新扩展，并将其插入到这个环中。
- Oracle 不会使用被尾部占据的扩展。

从上述原则可以看出，事务的持续时间和事务的大小都是影响回滚段的重要因素，一个长时间事务哪怕只使用了一个字节，也可能导致该扩展不能被马上重用，而造成回滚段扩展。

我们能够从这些原则中得到什么启发呢？在设置回滚段时，如果回滚段足够大，那么首先需要根据实际的事务大小来确定回滚段的存储参数，使回滚段头不会很快就转到段尾，以避免回滚段快速地进行扩展。第二个应该注意的问题是，如果执行一个长时间运行的查询，这个查询访问的数据变化频率十分快，那么就需要足够大的回滚段，以保证查询结束之前不会出现 ORA-1555 错误（SNAPSHOT TOO OLD）。

回滚段的大小取决于数据库中事务的特点，应该满足数据库日常事务的要求，而不是针对那些不经常使用的大事务。回滚段的数量设置要保证不会发生回滚段竞争。可以通过下列代码来查看是否存在回滚段竞争。

```
SELECT CLASS, COUNT FROM V$WAITSTAT WHERE CLASS = '%undo%';
```

任何非 0 的 COUNT 值都说明存在回滚段头争用。不过在 UNDO 自动管理的时代，这一切

往往不太需要DBA去操心了，只要UNDO表空间足够大，UNDO_RETENTION设置得合理一些，回滚段的争用就不会对系统造成多大的影响，ORA-1555这个世界级的难题也就迎刃而解了。不过如果SQL编写得太差，那么再大的UNDO_RETENTION也没办法避免ORA-1555，这时只能从应用优化的角度入手了。

事情总是有特例的，在某些情况下，使用UNDO自动管理，可能总是无法解决UNDO争用的问题，而且这种争用对系统总体性能影响较大，那么我们就必须放弃UNDO自动管理，改用UNDO手工管理了。尽管这种情况很少出现，老白还是遇到过的，因此对于广大DBA来说，虽然UNDO自动管理不需要你们辛苦地维护UNDO和回滚，但是了解一些UNDO手工管理的基本技术还是很有必要的。

6.1.2　ITL和UNDO

ITL就是我们常说的事务槽，它存在于数据块中，位于块头的44B之后。每个ITL项的大小都是固定的24B。在一个数据块被格式化后，首先会默认创建数个ITL（不同的版本略有不同，比如，10*g*版本的数据库表会默认创建两个ITL）。当ITL空间不够用时，系统会另外分配24B，扩展一个新的ITL。

ITL对于事务来说至关重要。在一个事务要修改某个数据块中的某条记录时，首先要找到一个空闲的ITL槽，然后将该ITL槽的序号登记在这个数据行的头部。ITL槽的结构如下：

```
struct ITL槽结构 {
 kxid_st  事务号 ;
 kuba_st  最后一个改动的UNDO地址（UBA）;
 ub2     锁标识，在锁定时记录本事务在本块中锁定的记录数
 ub2     SCN WRAP;
 ub4     SCN BASE;
} ;
```

不难看出，锁标识（ktbitflg）可以在事务锁定时记录被锁定的记录数，因此，同一个事务如果在一个数据块中修改了多条记录，只需要申请一个ITL槽。在事务提交之前，这个ITL槽是被锁定的，不能复用。事务提交或者回滚后，ITL槽解除锁定，其他事务可以再次使用这个ITL槽。如果块中的所有ITL槽都被占用了，那么就需要新扩展一个ITL槽供这个事务使用。这时如果不凑巧，块中已经无法再分配24B来扩充ITL槽了，那么这个事务就只能等待ITL。这种等待也会体现在TX行锁等待，不过这种行锁等待和普通的行锁等待不同，其LMODE是4而不是6。

如果ITL等待比较严重，那就需要加大INITRANS参数了。不过需要注意的是，加大这个参数后，新格式化的数据块会使用新参数，而旧的数据块不会改变，除非对表进行重组（比如MOVE

或者EXP/IMP）。

除了修改数据块的事务外，一致性读也依赖于ITL。如果某一致性读操作在读取某个数据块中的数据时，发现这个数据太新了，就会根据这条数据头部的ITC标识找到这个ITC通过ITL的UBA，并找到其修改前的前映像数据，从而读取到较早的数据。

通过上述内容，可能有些朋友的疑问已经得到了解答，我们也理解了一致性读是如何通过ITL找到前映像的。不过有些细心的朋友可能会思考，这样似乎还不足够，因为事务提交后，ITL可以被重用，那么以前的ITL信息可能会被覆盖，我们找到的数据就可能不是预期的。其实不用担心这个问题，在ITL被覆盖前，其信息会被复制到UNDO记录中，只要通过UNDO记录中原来的UBA信息，就可以找到正确的信息了。

6.1.3　如何转储UNDO

从Oracle 7.3开始，我们就可以转储UNDO的信息了。在日常维护工作中，转储UNDO信息的机会比较少。不过在一些故障分析或者性能分析中，有时还是会用到这项技术。这里将简单介绍如何转储UNDO信息。转储UNDO信息需要使用sysdba或者sysoper账号，然后通过ALTER SYSTEM DUMP UNDO命令来完成。其语法如下所示：

```
ALTER SYSTEM DUMP UNDO[ HEADER <rbs-name> ][ BLOCK <rbs-name> [ <filter> ] ];

Filters:
   XID <usn> <slot> <sqn>      //通过select XIDUSN,XIDSLOT,XIDSQN from v$transaction
                               //可以获取上述信息
   UBA MIN <file> <block>
   UBA MAX <file> <block>
   EXTENT MIN <extentno>
   EXTENT MAX <extentno>OBJNO <objno>
   LAYER <layer>
   LEVEL <level>
```

下面举例说明。

(1) 转储回滚段头。

(2) SQL> alter system dump undo header '_SYSSMU01$';

(3) 根据XID来转储UNDO链。

a. 找到XID。

```
SQL> select xidusn, xidslot, xidsqn from v$transaction;
  XIDUSN    XIDSLOT    XIDSQN
--------- ---------- ----------
    3          3        834
```

b. 根据上一步结果中的XIDUSN找到回滚段。

```
SQL> select name from v$rollname where usn=3;
NAME
------------------------------
_SYSSMU3$
```

c. 转储 UNDO 链。

```
SQL> alter system dump undo block '_SYSSMU3$' xid 3 3 834;
```

6.1.4 UNDO 自动管理是如何工作的

在前面已经提到，UNDO 自动管理可以将 DBA 从头疼的 UNDO 管理工作中解放出来。不过在某些情况下，我们可能需要调整 UNDO 自动管理的相关配置，甚至回过头来使用 UNDO 手工管理。因此通过本节的内容去了解 UNDO 自动管理中的一些基本原理还是十分必要的。

UNDO 的自动管理基于几个要素。首先，在 UNDO 自动管理模式下，一个数据库实例只能有一个独立的 UNDO 表空间，该实例中除 SYSTEM 以外的所有回滚段都会被创建在这个表空间中。

其次，回滚段是根据系统的负载情况动态创建的，处于不同的状态（ONLINE/OFFLINE）。SMON 根据平均每秒的事务数来确定系统中所需要的回滚段数量。每秒处理的事务数量越高，处于在线状态的（ONLINE）回滚段也就越多。如果业务高峰过去了，回滚段的需求量也会下降。这时如果有些回滚段的事务都已经提交了，而且所有的数据都已经不在 UNDO RETENTION 保护范围内，那么这些回滚段就会处于离线状态（OFFLINE）。

在 UNDO 自动管理的模式下，SMON 也会根据回滚段的使用情况，自动计算理想数量，确保在线的回滚段不会出现抖动现象。

如果我们使用 10*g* 版本的数据库，那么_UNDO_AUTOTUNE 参数默认为 TRUE。这种情况下，SMON 会根据 MAX QUERY LENGTH 和 UNDO 表空间的大小自动调整 UNDO_RETENTION 的值，尽量使最长的查询不出现 ORA-1555 错误。这种管理模式会尽可能地使用 UNDO 表空间，从而确保 MAX QUERY LENGTH。

Oracle 10*g* 如何实现 UNDO_RETENTION 的自动调整呢？如果将 UNDO_MANAGEMENT 设置为 AUTO，那么无论 UNDO_RETENTION 是否被设置，MMON 都会以 30 秒为周期，计算 MAX QUERY LENGTH，并将 TUNED_UNDORETENTION 参数设置为“MAX QUERY LENGTH + <修正值>”。这一点可以通过下面的查询来验证。

```
SQL> select tuned_undoretention, maxquerylen, maxqueryid from v$undostat;
TUNED_UNDORETENTION  MAXQUERYLEN  MAXQUERYID
-------------------- ----------- -------------
              66438        65534  dc78zsn04tj4u
              65829        64925  dc78zsn04tj4u
              65221        64316  dc78zsn04tj4u
              64673        64012  dc78zsn04tj4u
              64064        63404  dc78zsn04tj4u
              63456        62795  dc78zsn04tj4u
              62848        62187  dc78zsn04tj4u
              62239        61578  dc78zsn04tj4u
              61630        60969  dc78zsn04tj4u
              61022        60361  dc78zsn04tj4u
              60474        59753  dc78zsn04tj4u
```

某些文档中提到的修正参数是 300，但老白在不同系统中观察到的情况略有不同。不过这一点

已经不重要了,我们需要知道的是 MAX QUERY LENGTH 对 UNDO_RETENTION 管理产生的影响。

6.1.5　系统回滚段的作用

使用 CREATE DATABASE 命令创建数据库，仅仅会创建一个回滚段——系统回滚段，这个回滚段会被创建在 SYSTEM 表空间中。系统回滚段和其他回滚段（包括其他创建在系统表空间中的回滚段）存在本质的不同，系统回滚段只能用于对系统表空间的对象进行操作的事务。设置系统回滚段是为了给 Oracle 的一些内部事务提供回滚空间，包括一些 Oracle 空间管理操作和一些对系统的数据字典表进行的操作。

下列代码可以验证系统回滚段的使用范围。

```
--首先创建两张表，test1 存放于系统表空间，test2 存放于用户表空间。

SQL> create table test1 (a integer ) tablespace system;
Table created.

SQL> create table test2 (a integer) tablespace users;
Table created.

--确认只有系统 RBS 是在线的。

SQL> select segment_name ,status from dba_rollback_segs;
SEGMENT_NAME                   STATUS
SYSTEM                         ONLINE
_SYSSMU1$                      OFFLINE
_SYSSMU2$                      OFFLINE
_SYSSMU3$                      OFFLINE
_SYSSMU4$                      OFFLINE
_SYSSMU5$                      OFFLINE
_SYSSMU6$                      OFFLINE
_SYSSMU7$                      OFFLINE
_SYSSMU8$                      OFFLINE
_SYSSMU9$                      OFFLINE
_SYSSMU10$                     OFFLINE

SQL> insert into test1 values (1);
1 row created.
SQL> commit;
SQL> insert into test2 values (2);
insert into test2 values (2)
            *
ERROR at line 1:
ORA-01552: cannot use system rollback segment for non-system tablespace 'USERS'
```

从上面的 ORA-1552 报错可以看出，非系统表空间的对象不能使用系统 RBS。这样设置的缺点是，系统回滚段不能保证只被数据字典表使用。如果一张用户表被创建在系统表空间，那么该表的操作也可以使用系统表空间。因此不允许把用户表创建在系统表空间是一条铁律。

其他位于系统表空间的回滚段没有这种限制，但是仍然不建议在系统表空间中存放其他回滚段，以避免引起性能问题。

6.1.6 著名的 ORA-1555

如果我们要进行大查询（一般把执行时间比较长、开销比较大的查询称为大查询），比如进行一个大的统计分析，那么可能会出现 ORA-1555 错误 （SNAPSHOT TOO OLD）。从 Oracle 的错误信息来看，这个错误是回滚段太小导致的，其实这是一种误导。无论怎样设置回滚段的大小，都有可能发生这个问题，加大回滚段的大小并不能解决这个问题。

一个事务启动时，Oracle 会保存它启动时的 SCN。当该事务执行的时候，Oracle 会检查所有的行，确认这些数据从事务启动的时间点开始就没有发生变化。如果要访问的数据发生了变化，那么 Oracle 会到回滚段中去查找那个时间点的数据（通过比较 SCN 可以找到这个数据）。对于没有提交的事务，数据会一直在回滚段中存在，这些数据不会产生 ORA-1555 问题。而对于那些已经提交的事务，回滚段可能会被其他事务覆盖或者在使用 OPTIMAL 模式时被 Oracle 释放。碰到这种情况，就会出现 ORA-1555 错误。无论系统拥有多大的回滚段，提交过的事务相关的回滚段空间都有可能被覆盖。回滚段越大，被覆盖的机会越小，所以错误信息才会提示回滚段太小。但是机会小不等于不发生，因此简单地扩大回滚段并不是解决 ORA-1555 问题的有效方法。

6

解决 ORA-1555 问题最简单的方法是调整那些较大事务的执行时间，使之能够在其他事务较小时执行，或者把大事务分解为一些小事务。但事实上最简单的解决方法往往也是最不可能实现的方法，很多应用系统无法完全按照 DBA 的意志去执行。因此，要从技术角度去解决 ORA-1555 问题，这需要从以下几个方面入手。

首先，确保所有可用的回滚段都是在线的。回滚段越多，回滚信息被覆盖的机会就越少。

其次，确保所有回滚段的扩展属性都是相同大小的。因为最容易导致 ORA-1555 错误的回滚段是那些较小的、很容易发生回转的回滚段。

从 Oracle 9*i* 开始，设置 UNDO_RETENTION 参数可以解决大部分 ORA-1555 问题。UNDO_RETENTION 参数的含义是 UNDO 信息在 UNDO 表空间中保留的时间，单位是秒。要注意的是，在 RAC 环境下使用该参数，所有的实例都要设置相同的值，否则会导致数据库故障。设置较大的 UNDO_RETENTION 参数时，要注意 UNDO 表空间需要有足够的空间来保留旧数据。另外，UNDO_RETENTION 并不是任何时候都能得到保证的，也就是说，并不能保证所有的 UNDO 数据都不会在 UNDO_RETENTION 保护的时间内被覆盖，如果 UNDO 表空间的容量不足以支撑 UNDO_RETENTION，那么在 UNDO_RETENTION 保护范围内的 UNDO 数据也有可能被覆盖。从 Oracle 10*g* 开始提供了“保证”模式，确保 UNDO 记录在 UNDO_RETENTION 保护范围内不会被覆盖。在“保证”模式下，如果 UNDO 表空间不足以支撑 UNDO_RETENTION 的需要，系统会出现 UNDO 表空间不足的错误，因此要使用“保证”模式，就必须确保 UNDO 表空间充足。

虽然 Oracle 9*i* 提供了一些手段来解决 ORA-1555 问题，但对于已设计完成的应用系统，优化应用才是最重要的。如果某个操作在以前是可以完成的，而现在出现了 ORA-1555 错误，那么不要急于调整 UNDO_RETENTION 参数，而要尽量找出原因所在，是因为数据量的变化导致执行时间变长，还是因为优化器选择了非最优的执行计划……找到真正的故障并排除它才是最好的解决方案。

6.1.7　回滚段手工管理

一般情况下，现在我们很少会采用 UNDO 的手工管理，不过在一些极端的应用场合，比如，UNDO 自动管理碰到一些可能导致严重后果的 Bug，或者应用的极端性导致自动管理无法正常工作，这时就只能手工管理回滚段了。

在一个事务的大小和并发性波动都不大的系统中，配置回滚段表空间十分容易。通过前面几节介绍的方法，可以很容易地计算出系统中大致需要多少个回滚段，以及每个回滚段的最佳大小。创建一个表空间，使其空间不小于所有回滚段最佳大小总和的 120%（如果存储空间足够，建议不小于所有回滚段最佳大小的 150%，甚至更多）。如果会出现比较大的事务，可以适当调高这个比例。创建回滚段的时候，使用 OPTIMAL 参数，并且设置所有的回滚段都具有相同的扩展参数。

经常出现大型事务操作的系统是最难管理的系统，有两种方案可以选择，一种是创建较少的回滚段，每个回滚段的大小都比较大，能够适合大型事务。另一种方案是对于大型的事务处理，使用独立的大型回滚段。由于第一种方法减少了回滚段的数量，因此可能会导致回滚段段头的冲突。第二种方法需要应用程序使用 SET TRANSACTION USE ROLLBACK SEGMENT 命令来指定回滚段（需要注意的是，对于 DDL 操作，这条语句无效），并且需要对应用系统进行修改。在应用程序层面就很好地划分回滚段的使用虽然会加大开发的难度，但是系统往往可以获得最佳的性能。在这种情况下，OPTIMAL 参数就不适合使用了。

对于存在少量大事务的系统，比如，一个偶尔进行统计操作的 OLTP 系统，在这种环境下使用 OPTIMAL 参数，并设置每个回滚段的 OPTIMAL 值是最小的回滚段覆盖值，同时保证回滚段所在的表空间能够满足大型事务扩展回滚段的需求。在这样的配置下，某些回滚段会偶尔扩展到很大，但是很快这些回滚段又会恢复正常。这样会产生一些回滚段申请和释放带来的开销，导致性能略微下降。因此在这种情况下，选择合适的 OPTIMAL 大小十分关键。

在创建回滚段的语句中，可以使用 OPTIMAL 选项。使用该选项后，系统会通过自动管理来实现最佳的回滚段大小，从而使系统中回滚段的浪费和扩展最小化。使用 OPTIMAL 选项后，在事务提交后，如果不再使用回滚段的数据，超过 OPTIMAL 指定值的部分回滚段会被自动缩小。

当一个导致回滚段扩展的事务完成后，在回滚段头移动到下一个数据块时，系统会计算段的大小。如果超过了最优值，那么数据库会考虑释放一些扩展。如果在两个连续的扩展中都没有活跃的事务，就会释放其中一个扩展。需要保持两个连续的空扩展的唯一理由是，当前的事务可能需要使用其中的一个扩展。（如果空扩展被释放了，而下一个扩展正好又是满的，那么只能重新分配一个扩展。）如果需要的话，在一个事务中，可以释放多个扩展。由于回滚段采用的是循环使用机制，以这种方式释放的扩展所包含的数据是最旧的。

OPTIMAL 语句是一种十分有效的工具，在使用 OPTIMAL 时需要注意一些问题。首先，分配和释放扩展会带来较大的开销，如果 OPTIMAL 设置得不合理，就会带来性能问题，特别是在 OPTIMAL 过小时。在设置回滚段的大小时，最好的方法是设置所有回滚段的大小都能适合单个事务的大小，这看起来简单，但实施起来难度却很大，比如，系统最大的事务是 2GB，而系统中有几十个回滚段，要把每个回滚段都设置为 2GB 是很难实现的。因此，回滚段的最优设置应该

能够适合 90%左右的事务。

另外要注意的是，在事务结束时，不会马上回收回滚段。只有当另外一个事务转移到下一个扩展，发现需要回收空间时，才会进行回收。

最后一点老白想要说的是，在 UNDO 手工管理的时代，存储空间是十分昂贵的，DBA 往往无法获得足够的 UNDO 空间。因此我们在计算回滚段数量和 OPTIMAL SIZE 的时候往往会斤斤计较。在存储空间价格已经降低到每 TB 容量 1 万元左右的时候，大多数情况下我们没必要故意为难自己。设置较大的 UNDO 表空间总是不会有错的，一些那个时代的 UNDO 管理技巧也到了要抛弃的时候了。

6.2　如何分析和优化 UNDO

UNDO 优化在自动管理和手工管理模式下是截然不同的。在 UNDO 手工管理模式下，UNDO 优化具有一定的难度。首先，我们需要根据系统的并发度，考虑回滚段（RBS）的数量，创建足够的 RBS 是确保 UNDO 效率的重要因素。接下来，我们需要考虑的问题是每个 RBS 的 OPTIMAL SIZE，也就是最佳 RBS 大小。在 RBS 扩展到大于 OPTIMAL SIZE 的情况下，当 RBS 不再需要那么大的空间时，就会逐渐释放，直到收缩至 OPTIMAL SIZE 指定的大小。设置合理的 OPTIMAL SIZE 可以避免 RBS 不停地扩展和收缩，从而保证 RBS 的性能。

总体来说，在手工管理回滚段的时代，如果碰到了回滚段导致的性能问题，那么处理起来还是十分棘手的，DBA 必须对回滚段有深刻的认识才能够驾驭其优化工作。幸运的是，现在我们很少会遇到这样的问题了，在 UNDO 自动管理的情况下，回滚段出现性能问题的机率十分小。我们可以通过 US-Undo Segment 锁等待所占的比例来判断 UNDO 是否存在严重的性能问题。这个锁等待如果排在等待事件的前列，就说明 UNDO 自动管理出现了性能问题。一般情况下，我们只需设置足够大小的 UNDO 表空间，其他的事情就可以交给 SMON 去管理了。在 AWR 报告中，我们可以通过 UNDO 的统计信息来判断其健康性。

```
                    Num Undo    Number of Max Qry  Max Tx Tun Ret STO/    uS/uR/uU/
End Time              Blocks Transactions Len (s)   Concy  (mins) OOS     eS/eR/eU
------------ ----------- ------------ ------- ------- ------- ----- ------------
26-Mar 10:01      42,139       42,710  44,800      29   2,627 0/0   1/0/0/12/359
26-Mar 09:51      59,680       46,808   1,410      48   2,633 0/0   7/0/0/4/8576
26-Mar 09:41      53,822       46,611   2,841      25   2,640 0/0   0/0/0/0/0/0
26-Mar 09:31      51,879       46,734   2,238      40   2,640 0/0   3/0/0/7/2969
26-Mar 09:21      47,943       51,812   1,635      31   2,650 0/0   2/0/0/8/2252
26-Mar 09:11      43,589       56,175   2,202      42   2,720 0/0   0/0/0/11/192
26-Mar 09:01      42,703       53,051   1,599      30   2,800 0/0   0/0/0/12/168
```

其中的 STO 是指 SNAPSHOT TOO OLD，也就是著名的 ORA-1555 错误。如果这个值不为零，说明在某个时间段出现了 ORA-1555。OOS 是指出现 OUT OF SPACE 的计数，如果这个值不为零，说明 UNDO 表空间不足。Max Qry Len (s)是指 MAX QUERY LENGTH，单位是秒。Max TX Concy 是指最大并发事务数，Tun Ret 是指自动调整的 RETENTION 值。uS/uR/uU/eS/eR/eU 这几项指标对于分析 UNDO 的使用情况也是十分有价值的。

- uS——unexpired Stolen；
- uR——unexpired Released；
- uU——unexpired reUsed；
- eS——expired Stolen；
- eR——expired Released；
- eU——expired reUsed。

如果 uS、uR 和 uU 的值经常不为零，那么说明 UNDO 表空间可能不足。UNDO SEGMENT 在某个时间点上无法扩展，只能强制回收还没有过期的 EXTENT，来确保新的事务能够正常运行。

在现在的 10*g*、11*g* 环境下，只要 UNDO 表空间的 I/O 性能能够得到保证，在 UNDO 表空间足够大的情况下，一般来说，UNDO 不会出现影响系统性能的重大故障。如果出现了 UNDO 导致的性能故障，那么要首先观察这是否是由于 UNDO SEGMENT OFFLINE/ONLINE 十分频繁所导致的。如果是这样，就说明系统可能出现了 Bug，我们可以到 MOS 上查找相关的信息，或者干脆开个 SR。但如果是由于大量并发事务所导致的，那么最好还是从应用的角度去分析，看看为什么会有这么大的并发事务，能否通过优化应用来解决。而在最为极端的情况下，我们就只能放弃 UNDO 自动管理，采用手工管理 UNDO 的方法来解决这个问题了。

第 7 章

理解 PGA、临时表空间和排序

PGA 是 Process Global Area 或者 Program Global Area 的缩写，指服务进程的私有内存空间。PGA 中包含服务进程的全局变量、数据结构和控制信息，比如服务进程执行查询时的游标。PGA 包含的信息如下所示。

- 私有 SQL 区域：存储服务进程执行 SQL 所需要的私有数据和控制结构，包括固定区域和运行时区域（Run-time Area）。固定区域的数据在游标关闭前一直存在，运行时区域在 SQL 执行的时候存在。Oracle 执行 SQL 的第一步就是创建运行时区域，对于 INSERT、DELETE、UPDATE 语句，执行结束后，该区域就被释放了；而对于 SELECT 操作，只有所有的结果集都被读取完毕或者查询取消了，该区域才会被释放。对于独立服务器模式的连接，私有 SQL 区域在 PGA 中分配空间；对于共享服务器模式的连接，私有 SQL 区域在 SGA 中分配空间。
- 会话空间：用于存放 logon 信息等会话相关的控制信息。对于共享服务器模式，会话空间是共享的。
- SQL 工作区域（SQL Work Area）：对于复杂的 SQL，比如多表连接的统计查询，在 SQL 执行过程中，如果有下列操作，则会使用一些额外的区域。
  - 排序操作，当 SQL 中有 order by、group by、rollup 等操作的时候，需要使用排序空间来处理。
  - 多表的 HASH 连接，需要使用 Hash Join Area 来处理。
  - 位图连接。
  - 创建位图。

PGA 使用的空间是独立于 SGA 的。PGA 中的 SQL 工作区域对于系统的性能影响很大，配置合理的 SQL 工作区域参数对于调整系统性能具有十分重要的作用。

管理 PGA 也是 DBA 一项十分重要的工作，设置合理的 PGA 可以减少硬盘排序的数量，提高大型表连接和统计操作的性能。要想了解 PGA，我们就要先从临时表空间和临时段开始。

7.1　基本概念

尽管 PGA 比较易于理解，但仍然有很多 DBA 理解得不够透彻。因为 PGA 的构成非常复杂，不像 SGA 中的组件那么清晰明了。PGA 包含了进程中私有的内存，这些内存的功能十分广泛。本节将介绍 PGA 的基本概念，主要集中在 PGA 和 SQL 执行相关的内容，包括排序区、HASH 工作区等各类工作区。

当然，提到排序，就免不了谈到临时段和临时表空间，因此本节并未从 PGA 本身谈起，而是先从临时段和临时表空间开始。

7.1.1　临时表空间和临时段

临时表空间操作和排序操作是数据库中最常见的操作，也是令绝大多数 DBA 最为困惑的操作。本节就是要向 DBA 介绍临时表空间，以及在临时表空间中进行的主要操作。

临时表空间用于存放排序、临时表等数据，其信息不需要 REDO，因此临时表的 DML 操作往往比普通表低得多。临时段不仅仅存在于临时表空间中，也可能存在于普通的表空间里。比如，我们通过 CTAS 创建一张表，在 CTAS 命令没有结束前，新表的数据是放在临时段中的，这些临时段在 CTAS 完成的时候会被转换为 PERMENT 段。

对于临时表空间的管理机制和临时段的算法，Oracle 7.3 是一个革命性的版本。由于 7.3 版本推出了 Oracle 第一个集群软件 OPS，而原有的临时段算法无法适应 OPS 的需要，因此 Oracle 又推出了一种全新的临时段管理算法。在 7.3 版本之前，临时段是在需要时分配，使用完毕后就被删除的。Oracle 7.3 推出的新算法的核心就是 SEP（SORT EXTENT POOL），SEP 负责管理临时段中扩展的结构，存储在共享池内，任何需要使用排序空间的操作，都需要从 SEP 中分配空闲的扩展，使用完毕后，不需要释放该空间，只需要在 SEP 中将该扩展设置为空闲。

当数据库实例启动后，SMON 将会删除该实例未释放的临时段，并且对临时表空间进行碎片整理。在这个操作完成前，数据库打开的操作不能完成。因此每次数据库重启后，临时段中的垃圾都会被完全清理。当数据库打开后，第一个进行的硬盘排序操作会在相关的临时表空间内创建临时段，这个临时段也是整个实例唯一的临时段（在新的临时段算法下，同一个表空间内，每个实例只有一个临时段）。临时段中扩展的信息会被记录在 SEP 中。硬盘排序操作会在 SEP 中查找可用的扩展，在查找前，需要获得 SORT EXTENT POOL 闩锁。如果能找到可用的扩展，那么 SEP 中已被分配的扩展就会被标注为占用状态；如果找不到可用的扩展，那么系统就会试图从表空间中分配新的空间，而如果这个分配工作因为表空间的空闲空间不足而无法完成，那么就会产生一个 ORA-1652 错误。

当排序操作完成的时候，会再次获取 SORT EXTENT POOL 闩锁，并且将使用的扩展标注为空闲，然后释放 SORT EXTENT POOL 闩锁。

新的临时表空间管理算法不需要频繁地分配和释放临时段，这大大提高了临时段管理的效率。由于这是一种只分配不释放的算法，因此 DBA 经常会看到自己的临时表空间总是处于或者

接近 100%使用的状态。其实对于 7.3 以后的版本而言，临时表空间使用率接近 100%是十分正常的，DBA 可以通过 V$SORT_USAGE 和 V$SORT_SEGMENTS 这两个视图来检查临时段的使用情况。

在 RAC/OPS 环境下，临时段的管理算法也是类似的。在 RAC 中，由于多个实例会共享一个临时表空间，因此这些实例也能够共享相同的扩展。在 RAC 环境下，每个实例都拥有独立的 SEP，各个实例中的排序操作需要在自己的 SEP 中分配空间。如果 SEP 中无法分配到足够的空间，那么首先会在表空间中分配；而如果表空间也已经分配完毕，这时若其他的实例还有空闲的扩展，那么这个扩展就可以分配给需要的实例使用。这些操作都是不可见的，虽然 SERVER 进程不会收到 ORA-1652 的错误信息，但是在 ALTER LOG 中会有一个 ORA-1652 的记录。

7.1.2　PGA 和排序

PGA 使用的空间是独立于 SGA 的，主要用来进行排序、表连接等操作。一般来说，根据系统配置和应用软件的不同，可以把物理内存的 15% ~ 30%用于 PGA 空间。PGA 中的 SQL 工作区域对于系统的性能影响很大，配置合理的 SQL 工作区域参数对于调整系统性能具有十分重要的作用。Oracle 9*i* 提供的自动 PGA 内存管理功能，可以帮助我们更加灵活地使用和管理 PGA。

大多数 SQL 操作都与数据排序相关，特别是对数据的统计和分析。排序和数据库的性能关系密切。在下列情况下会发生排序操作。

- 创建索引：CREATE INDEX 语句会引起服务进程（对于并行操作，是从属进程在进行操作）对索引字段进行排序，并创建 B 树。当排序结束的时候，会在索引的目的表空间中创建临时段来存放结果，在索引创建完毕后，临时段会被转为索引段。
- 在 SELECT 语句中的 order by 或者 group by 子句。
- 在 SELECT 语句中使用 distinct 关键字。
- 在查询中使用 UNION、INTERSECT 或者 MINUS 操作，这些操作会导致服务进程查找重复的记录。
- SORT-MERGE 连接：如果在等于连接中缺乏合适的索引，那么连接操作会产生一个全表扫描操作，并且通过排序后的合并操作来完成连接操作。
- ANALYZE 语句：ANALYZE 语句执行时会排序数据，用以生成汇总数据。
- 其他的一些操作也可能产生排序操作，比如 create primary key、enable constraint、create table、create table as select 等。

排序操作需要分配内存空间，这部分内存空间称为 SORT_AREA。在使用 PGA 自动管理的系统中，SORT_AREA 被自动管理，否则，由 SORT_AREA_SIZE 参数限定每个会话使用 SORT_AREA 的最大数量（单位是字节）。

在 PGA 手工管理的模式下，对于每个会话而言，在排序发生时，会立即分配一部分内存，随着排序操作的进行，内存会不断增长，直到排序操作完成或者达到本会话能够使用的排序空间的最大值。当排序完成后，如果使用的排序内存空间大于 SORT_AREA_RETAINED_SIZE 参数

指定的大小，那么部分空间会被释放，而由这个参数指定了大小的内存空间会被会话保留，以便用于下次排序。当服务进程退出系统的时候，排序空间被完全释放，并归还给系统。

当为排序分配的空间已经达到了 SORT_AREA_SIZE 参数指定的大小，而排序操作仍未完成，还需要继续使用排序空间时，那么已经排序的行会被写入磁盘，然后释放使用过的排序空间，供后面还没有完成的排序操作使用。服务进程会在临时表空间中创建临时段来存放这些数据，今后对于这些行的排序操作，需要访问磁盘。这些排序使用的临时段一般存储在临时表空间中，也可以存储在永久性表空间中。

排序使用的临时段会在数据库发生第一次排序操作时，在临时表空间中被创建。多个需要在磁盘中排序的事务可以共享同一个排序段，但是它们不使用相同的扩展（extent）。排序使用的扩展在实例关闭以前不会自动释放，但是会被标注为空闲状态，以便其他排序操作使用。因此，在数据库启动后，临时表空间的空闲空间会持续下降。Oracle 数据库把排序段的信息存储在 SEP（Sort Extent Pool，排序扩展池）中，SEP 是 SGA 中的一种控制结构。每个需要使用临时表空间进行排序的 SQL 语句都会检查 SEP，去查找可使用的空闲扩展。后台进程 SMON 会在数据库实例启动的时候，在数据库被打开后释放所有的排序段，这就是每次数据库启动后，SMON 会占用大量 CPU 资源的原因。这种操作可以避免由临时表空间的碎片化所导致的性能问题。

基于 Oracle 排序操作的特点，对于临时表空间的设置，提出如下的建议。

- 在系统中使用多个临时表空间，以避免数据访问冲突。在 Oracle 10*g* 或者 11*g* 版本中，可以使用排序表空间组。
- 把临时表空间的文件分散到多个磁盘上，以提高排序操作的性能。
- 对于复杂的系统，可以创建多个存储参数完全不同的临时表空间，供不同特性的排序操作使用。
- 在 Oracle 9*i* 及以后的版本中，使用本地管理表空间作为临时表空间。在 Oracle 9*i* 以前的版本中，临时表空间的存储参数设置为 INITIAL=NEXT，因为每次写入磁盘的数据量基本上是一致的（等于 SORT_AREA_SIZE，在自动 PGA 管理中，等于系统允许会话使用的 SQL 工作区的最大值）。可以这样计算 INITIAL 的值，等于或者略大于 $n \times$ SORT_AREA_SIZE + DB_BLOCK_SIZE。采用这种设置，每个扩展都可以存放一次排序操作所需要的空间（这只是一种建议的设置，在大多数情况下可以满足系统的要求，但并不一定在所有的场合下都会提高系统的性能）。
- 如果一张表是按照某个键值的升序装载的，那么在创建索引的时候可以使用 NOSORT 子句来避免不必要的排序。
- 对于并行查询，每个查询会使用比普通查询更多的排序区资源。
- 由于磁盘排序的效率比内存排序低很多（慢 100 倍以上），因此需设置合理的 SQL 排序区大小，并尽量避免在排序中出现磁盘操作。对于 OLTP 系统，设置足够大的 PGA 空间可以使系统的内存排序比例超过 99%。

从 V$SORT_SEGMENT 中可以查看到临时表空间中排序段的情况。通过下列代码所示的 SQL 语句可以查看当前临时段的使用情况：

```
SELECT tablespace_name, extent_size, total_extents, used_extents,
         free_extents, max_used_size
      FROM v$sort_segment;
```

当排序正在进行时，我们可以看到下面的结果：

```
TABLESPACE_NA EXTENT_SIZ TOTAL_EXTE USED_EXTEN FREE_EXTEN MAX_USED_S
------------- ---------- ---------- ---------- ---------- ----------
TEMP01                 9         25         13         12         20
1 row selected.
```

排序结束后，结果如下：

```
TABLESPACE_NA EXTENT_SIZ TOTAL_EXTE USED_EXTEN FREE_EXTEN MAX_USED_S
------------- ---------- ---------- ---------- ---------- ----------
TEMP01                 9         25          0         25         20
1 row selected.
```

可以看出，排序后，TEMP01的段中共有25个扩展，当排序进行的时候使用了13个，剩余12个空闲扩展，当排序结束后，25个扩展都处于空闲的状态。如果要查看当前的某个数据库用户使用了多少临时段空间，可以通过查询V$SORT_USAGE和V$SESSION实现：


```
SELECT s.username, u.tablespace, u.contents, u.extents, u.blocks
      FROM v$session s, v$sort_usage u
      WHERE s.saddr=u.session_addr;
```

如果test用户正在做一个排序，那么会出现如下结果：

```
USERNAME TABLESPACE CONTENTS  EXTENTS    BLOCKS
-------- ---------- --------- ---------- ----------
TEST     TEMP01     TEMPORARY         13        130
1 row selected.
```

当排序结束的时候，V$SORT_USAGE中的数据将被清除，分配的扩展也就归还了。

7.1.3　PGA和PGA_AGGREGATE_TARGET

在Oracle数据库中，和PGA相关的参数包括：

- PGA_AGGREGATE_TARGET（9*i*+），PGA能够使用的最大内存空间；
- WORKAREA_SIZE_POLICY（9*i*+），分配SQL工作区域的规则，可以取值AUTO或者MANUAL；
- SORT_AREA_SIZE，定义每个会话可以用于内存排序的空间的最大值；
- HASH_AREA_SIZE，定义每个会话可以用于哈希连接的内存空间的最大值；
- BITMAP_MERGE_AREA_SIZE，定义每个会话使用位图合并连接时的内存工作区域的最大值；
- CREATE_BITMAP_AREA_SIZE，定义每个会话创建位图时可以使用的内存工作区域的最大值。

可以根据实际应用的需要，来设置PGA_AGGREGATE_TARGET参数。对于OLTP系统，一般不超过物理内存的20%。但这个规则不是绝对的，具体的值可以根据AWR或者STATSPACK

报告的建议来设置。如果将 WORKAREA_SIZE_POLICY 设置为 AUTO（Oracle 9*i* 或以后的版本才支持），那么就不需要设置*_AREA_SIZE 参数了，Oracle 会自动管理。（在 Oracle 9*i* 版本中，PGA 自动管理仅对独立服务器模式的会话有效，共享服务器模式的会话还需要使用*_AREA_SIZE 参数；从 Oracle 10*g* 版本开始，PGA 自动管理对两种模式的会话都有效。）

对于大型的查询操作，其性能取决于 PGA 的各个工作区域的设置。总的来说，大的 SQL 工作区域会提高查询的性能，不过会消耗比较大的内存。当 SQL 工作区域充足的时候，所有的相关操作都可以在内存中完成。当 SQL 工作区域不足的时候，需要在临时表空间中分配空间来完成操作，因此性能会受到很大的影响。在内存资源不是很充足的系统中，调整 SQL 工作区域是一项极具挑战的工作。

在 Oracle 9*i* 版本以前的数据库中，管理 PGA 是十分复杂的工作。需要 DBA 非常谨慎地配置各个 AREA_SIZE 参数。而 Oracle 9*i* 提供了一种全自动管理 PGA 的方法，只要把 WORKAREA_SIZE_POLICY 参数设置为 AUTO，并且设置一个合适的 PGA_AGGREGATE_TARGET 值，Oracle 数据库就会自动管理 PGA 的各个工作区域。

PGA_AGGREGATE_TARGET 参数值的设置并不是按照完全固定的方法来完成的。MOS 上的一篇官方文档提出了一种具有一定普遍性的设置方式：对于 OLTP 系统，设置该值为物理内存的 15%～20%；而对于 OLAP 系统，则设置为物理内存的 40%。实际上，对于具体的系统来说，设置该参数并不是那么简单的，要根据当前系统的现状来设置才有针对性。

通过 V$PGASTAT 可以得到一些调整 PGA_AGGREGATE_TARGET 参数的参考资料。比如：

```
SQL>SELECT * FROM V$PGASTAT;

NAME                                        VALUE
------------------------------------------- ------------
aggregate PGA target parameter              524288000 bytes
aggregate PGA auto target                   463435776 bytes
global memory bound                         25600 bytes
total PGA inuse                             9353216 bytes
total PGA allocated                         73516032 bytes
maximum PGA allocated                       698371072 bytes
total PGA used for auto workareas           0 bytes
maximum PGA used for auto workareas         560744448 bytes
total PGA used for manual workareas         0 bytes
maximum PGA used for manual workareas       0 bytes
over allocation count                       0 bytes
total bytes processed                       4.0072E+10 bytes
total extra bytes read/written              3.1517E+10 bytes
cache hit percentage                        55.97 percent
```

主要的状态信息包括如下几项。

- aggregate PGA auto target：可调整的 PGA 空间，只有这部分空间可以被 Oracle 自动调整功能使用，并用于各种 SQL 工作区域。这部分空间应该占 PGA_AGGREGATE_TARGET 中的最大比重，如果太小，会引起性能问题。
- total PGA used for auto workarea：系统使用的可调整的 PGA 空间，maximum PGA used for

auto workareas 显示系统启动后使用可调整 PGA 的最大值。

- total PGA in used：目前 PGA 空间的使用情况，这个值和 V$PROCESS 中的 PGA_USED_MEM 相同。
- over allocation count：这个值只在 Oracle 9*i* R2 版本中出现，如果 PGA_AGGREGATE_TARGET 的值设置得过小，Oracle 会分配额外的空间用于 PGA，这个值将会大于 0，此时应该调整 PGA_AGGREGATE_TARGET 参数，使这个值为 0 或者接近于 0。
- cache hit percentage：这个值只在 Oracle 9*i* R2 版本中出现，它指明了 PGA 的性能。如果这个值是 100%，说明 PGA 空间是足够的，都可以用优化模式（optimal）来完成操作。当 PGA 空间不足时，Oracle 需要使用外部辅助空间，结合 One-Pass 或 Multi-Pass 模式来完成操作，此时系统的性能将受到影响，该值的计算公式为：

```
                                           total bytes processed * 100
PGA Cache Hit Ratio =  ------------------------------------------------------
                      (total bytes processed + total extra bytes read/written)
```


使用 V$SQL_WORKAREA_HISTOGRAM 视图可以查看系统中 PGA 的详细使用细情况，该视图中列出了工作区域使用优化模式、One-Pass 模式和 Multi-Pass 模式的统计值，比如：

```
SELECT LOW_OPTIMAL_SIZE/1024 low_kb,(HIGH_OPTIMAL_SIZE+1)/1024 high_kb,
       optimal_executions, onepass_executions, multipasses_executions
FROM   v$sql_workarea_histogram
WHERE  total_executions != 0;

LOW_KB HIGH_KB OPTIMAL_EXECUTIONS ONEPASS_EXECUTIONS MULTIPASSES_EXECUTIONS
------ ------- ------------------ ------------------ ----------------------
8      16      156255             0                   0
16     32      150                0                   0
32     64      89                 0                   0
64     128     13                 0                   0
128    256     60                 0                   0
256    512     8                  0                   0
512    1024    657                0                   0
1024   2048    551                16                  0
2048   4096    538                26                  0
4096   8192    243                28                  0
8192   16384   137                35                  0
16384  32768   45                 107                 0
32768  65536   0                  153                 0
65536  131072  0                  73                  0
131072 262144  0                  44                  0
262144 524288  0                  22                  0
```

从上面的数据可以看出，使用 8 ~ 1024 KB 空间的所有操作都是优化模式的，1024 ~ 2048 KB 的空间中有 551 次操作是优化模式的，16 次操作是 One-Pass 模式的，没有 Multi-Pass 模式的操作。另外，我们在 STATSPACK 报告中也可以看到这个查询的数据，不过 STATSPACK 报告中的数据只是两个 SNAPSHOT 之间的统计信息，而这个视图中的数据是从系统启动到目前的统计值。

通过这个视图也可以计算各种访问方式的百分比，如下列代码所示。

```
SELECT optimal_count, round(optimal_count*100/total, 2) optimal_perc,
       onepass_count, round(onepass_count*100/total, 2) onepass_perc,
       multipass_count, round(multipass_count*100/total, 2) multipass_perc
FROM
       (SELECT decode(sum(total_executions), 0, 1, sum(total_executions)) total,
               sum(OPTIMAL_EXECUTIONS) optimal_count,
               sum(ONEPASS_EXECUTIONS) onepass_count,
               sum(MULTIPASSES_EXECUTIONS) multipass_count
        FROM   v$sql_workarea_histogram
        WHERE  low_optimal_size > &nk*1024);   -- nk 是参数
```

通过 V$SQL_WORKAREA_ACTIVE 视图，可以查看到当前活跃的 PGA 工作区。

```
SELECT to_number(decode(SID, 65535, NULL, SID)) sid,
       operation_type OPERATION,trunc(EXPECTED_SIZE/1024) ESIZE,
       trunc(ACTUAL_MEM_USED/1024) MEM, trunc(MAX_MEM_USED/1024) "MAX MEM",
       NUMBER_PASSES PASS, trunc(TEMPSEG_SIZE/1024) TSIZE
FROM V$SQL_WORKAREA_ACTIVE
ORDER BY 1,2;

SID OPERATION         ESIZE     MEM       MAX MEM    PASS TSIZE
--- ----------------- --------- --------- --------- ----- -------
8   GROUP BY (SORT)   315       280       904         0
8   HASH-JOIN         2995      2377      2430        1   20000
9   GROUP BY (SORT)   34300     22688     22688       0
11  HASH-JOIN         18044     54482     54482       0
12  HASH-JOIN         18044     11406     21406       1   120000
```

从上例可以看出，SID 为 8 的会话在进行 HASH 连接，采用的是 One-Pass 模式，使用了 2377 KB 的 PGA 内存，最大使用了 2430 KB 的 PGA 内存，并且使用了 20000 KB 的临时表空间。ESIZE 字段显示的是 PGA 内存管理器允许该会话使用的最大工作空间的大小。当 SQL 语句执行完毕后，工作区域就会自动被释放。

工作区域的大小取决于 PGA_AGGREGATE_TARGET 参数的值。Oracle 9*i* R2 版本还提供了 V$PGA_TARGET_ADVICE 和 V$PGA_TARGET_ADVICE_HISTOGRAM 视图，可以帮助用户来选择合适的 PGA_AGGREGATE_TARGET 值。当 STATISTICS_LEVEL 设置为 TYPICAL 或者 ALL 的时候，Oracle 数据库会生成这两个视图的数据。相关查询示例如下：

```
SELECT round(PGA_TARGET_FOR_ESTIMATE/1024/1024) target_mb,
       ESTD_PGA_CACHE_HIT_PERCENTAGE cache_hit_perc,
       ESTD_OVERALLOC_COUNT
FROM   v$pga_target_advice;


TARGET_MB  CACHE_HIT_PERC ESTD_OVERALLOC_COUNT
---------- -------------- --------------------
63         23             367
125        24             30
250        30             3
375        39             0
500        58             0
600        59             0
```

```
700         59             0
800         60             0
900         60             0
1000        61             0
1500        67             0
2000        76             0
3000        83             0
4000        85             0
```

这个查询的结果列出了当 PGA_AGGRATE_TARGET 的取值为 TARGET_MB 时，PGA 的命中率和 OVERALLOC 的值。选择一个 OVERALLOC 尽量接近于 0，命中率尽量接近于 100%，并且系统内存容量足以支持的合理值作为 PGA_AGGREGATE_TARGET 的参数值。

使用 PGA 自动管理还应该注意以下问题。

- 当设置 PGA_AGGREGATE_TARGET 和 WORKAREA_SIZE_POLICY=AUTO 后，所有的*_AREA_SIZE 参数都会被系统忽略（仅针对独立服务器模式，在 Oracle 11*g* 版本之前，共享服务器模式下只能使用 PGA 手动管理）。
- 对于 Oracle 8*i* 或者更早的版本，PGA 空间是静态的，当进程启动并且分配了 PGA 空间后，这些空间不会自动释放，除非会话结束。如果操作系统内存不足，可能会导致系统的换页操作。因此，将*_AREA_SIZE 参数设置得过大会导致大量内存被使用。在 Oracle 9*i* 版本使用自动 PGA 管理后，Oracle 会将进程分配的 PGA 空间中目前不使用的空间回收，并分配给其他进程使用，这样可以提高工作区空间的使用率，减少内存浪费。
- 使用自动 PGA 内存管理机制可以限制每个 Oracle 进程使用的总内存的数量，从而使内存的使用率更高。
- 由于能够动态分配工作区空间，因此可以大幅减少 ORA-4030 错误的发生。
- 如果 ESTD_OVERALLOCATION_COUNT（V$PGA_TARGET_ADVICE_VIEW 中的字段）的值非 0，说明 PGA_AGGREGATE_TARGET 的值太小。
- 如果 PGA_AGGREGATE_TARGET 的值过小，会导致 SQL*Loader 数据装载速度大幅度减慢，因此部分用户反映，Oracle 9*i* 中 SQL*Loader 的速度比 8*i* 慢，这主要是因为没有合理地设置 PGA_AGGREGATE_TARGET 参数。
- VMS 操作系统不支持 PGA 自动管理。
- 在 HP-UX 11.0 下设置 PGA 自动管理会导致操作系统出现错误（PANIC），这是由 bug:2122307 引起的。

7.1.4 你应该知道的 PGA 自动管理内幕

在 Oracle 9*i* 之前，设置合理的*_AREA_SIZE 是 DBA 的重要职责，这也是最具有挑战性的工作之一。Oracle 9*i* 的自动 PGA 管理，可以减轻 DBA 管理*_AREA_SIZE 的工作。Oracle 根据一定的规则为每个会话自动分配 SQL 的工作区域空间。

Oracle 进行 PGA 自动管理时有一项基本的原则，每个会话分配到的 SQL 工作区域空间不超过 PGA_AGGREGATE_TARGET 参数指定空间的 5%。对于并行操作，Oracle 为每个会话分配的

SQL 工作区域不超过 PGA_AGGREGATE_TARGET 参数指定空间的 30%。而对于并行处理的操作，每个子作业占用的 SQL 工作区域不超过“(PGA_AGGREGATE_TARGET 指定空间 × 30%)÷ 并行度”。

实际上，每个会话可分配到的 PGA 空间还受一个隐含参数的限制，这个参数就是 _pga_max_size，其默认值是 200 MB，如果我们想要使用更大的 PGA 内存，那么就需要加大这个参数。不过，在一个经常使用并行查询的系统中加大_pga_max_size 参数是存在风险的，需要确保物理内存足够支撑大量的 PGA 需求，否则系统就会出现由 ORA-4030 错误导致的问题。

很多 DBA 对 PGA_AGGREGATE_TARGET 参数存在误解，认为这个参数设定的是分配给 PGA 的空间大小。实际上并不是这样的，PGA_AGGREGATE_TARGET 参数只是设定了一个 PGA 分配时用于计算 PGA 的参考值。在 SQL 执行计划分析的时候，如果需要产生排序，那么系统会根据 PGA_AGGREGATE_TARGET 的值估算排序区的空间，并且完成排序区空间的分配工作。如果在 SQL 的执行过程中，分配的 PGA 排序区出现不足，那么系统就会进行硬盘排序，而不会重新从 PGA 中再度分配空间。因此我们经常会看到 PGA 使用率很低，但是有很多小型排序也使用了 One-Pass 模式，相关示例如表 7-1 所示。

表　7-1

	PGA Aggr Target(M)	Auto PGA Target(M)	PGA Mem Alloc(M)	W/A PGA Used(M)	%PGA W/A Mem	%Auto W/A Mem	%Man W/A Mem	Global Mem Bound(K)
B	20 480	17 136	1 669.67	0.00	0.00	0.00	0.00	10 240 000
E	20 480	17 114	1 702.74	3.00	0.18	100.00	0.00	10 240 000

从上表可以看出，PGA_AGGREGATE_TARGET 参数设置为了 20 GB，根据 BEGIN 和 END 两个采样点的系统中的会话情况，计算出来的 AUTO PGA 的大小分别为 17136 MB 和 17114 MB，这个值没有达到 PGA_AGGREGATE_TARGET 的设定值，实际分配的 PGA 内存是 1669 MB 和 1702 MB，均小于估算的 PGA 大小。下面我们来分析表 7-2 所示的数据。

表　7-2

Low Optimal	High Optimal	Total Execs	Optimal Execs	1-Pass Execs	M-Pass Execs
2K	4K	179 830	179 830	0	0
64K	128K	810	810	0	0
128K	256K	372	372	0	0
256K	512K	517	517	0	0
512K	1024K	23 191	23 191	0	0
1M	2M	686	686	0	0
2M	4M	590	247	343	0
4M	8M	23	19	4	0
8M	16M	9	9	0	0
16M	32M	5	1	4	0
512M	1024M	74	0	74	0

从上表可以看出，有 343 次 2 ~ 4 MB 的小排序也使用了 1-PASS 排序。而 8 ~ 16 MB 的略大排序反而是在内存中完成的。经过分析，我们发现系统中存在一些 connect by 树状查询，对于这类查询，Oracle 优化器在评估排序区大小的时候，往往会出现偏差，如果预先分配的缓冲区略小，那么就会出现大量的硬盘排序。在这个案例中，我们可以将 PGA_AGGREGATE_TARGET 从 20 GB 加大到 30 GB，这样就大幅缓解了这个问题。

7.2　PGA 优化的要点

在目前 PGA 自动管理的模式下，PGA 优化相对较为简单，只要设置合理的 PGA_AGGREGATE_TARGET 参数值，基本就能够达到目的。一般来说，PGA 优化的要点就是在物理内存足够的情况下，尽可能多地使用 PGA 内存，减少硬盘排序，如图 7-1 所示。

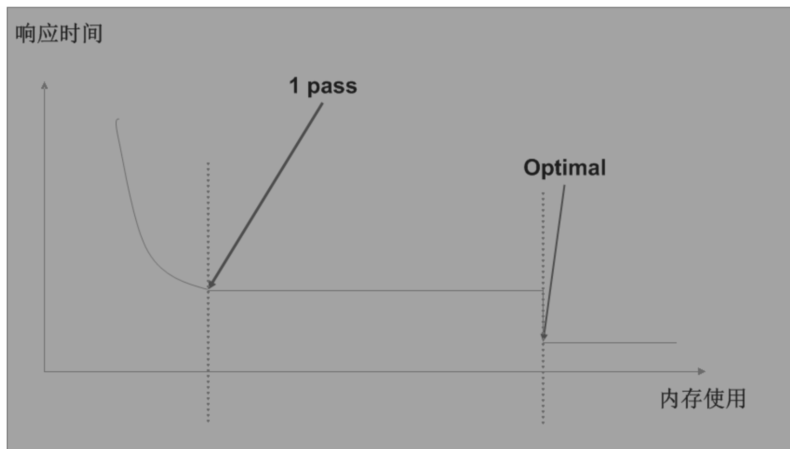


图　7-1

优化排序使用较多的物理内存，但是有最好的响应时间。我们可以通过 AWR 报告或者 STATSPACK 报告来判断 PGA 是否存在问题，相关示例如图 7-2 所示。

```
    % Blocks changed per Read:    8.94    Recursive Call %:    37.65
   Rollback per transaction %:    0.81       Rows per Sort:   122.30

 Instance Efficiency Percentages (Target 100%)
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

             Buffer Nowait %:   99.84       Redo NoWait %:   99.98
             Buffer  Hit   %:   99.46    In-memory Sort %:  100.00
             Library Hit   %:  100.05        Soft Parse %:  100.00
          Execute to Parse %:   60.22         Latch Hit %:   99.43
 Parse CPU to Parse Elapsd %:    9.02     % Non-Parse CPU:   99.94
```

图　7-2

如果 In-memory Sort 的指标小于 100%，就说明 PGA 存在一定的硬盘排序。下面我们根据如下 PGA 命中率进一步分析。

```
PGA Aggr Summary                    DB/Inst: SFOSS/sfoss2  Snaps: 12838-12839
-> PGA cache hit % - percentage of W/A (WorkArea) data processed only in-memory

PGA Cache Hit %   W/A MB Processed  Extra W/A MB Read/Written
--------------- ------------------ --------------------------
           98.4            302,582                      4,894
----------------------------------------------------------
```

我们可以看到 PGA 的命中率是 98.4%，这个比例算是比较高了。但命中率高并不等于没有问题，在理想的情况下，可以让 PGA 的命中率达到 100%，实现完全的内存排序，以获得最佳的性能。接下来，我们可以通过如下所示的 PGA 排序柱状图来判断是否存在问题。

```
  Low     High
Optimal Optimal    Total Execs  Optimal Execs 1-Pass Execs M-Pass Execs
------- ------- -------------- -------------- ------------ ------------
     2K      4K      2,097,268      2,097,268            0            0
    64K    128K          7,485          7,485            0            0
   128K    256K          5,214          5,214            0            0
   256K    512K          4,755          4,755            0            0
   512K   1024K        394,849        394,839           10            0
     1M      2M          4,695          4,695            0            0
     2M      4M            776            538          238            0
     4M      8M            317            139          178            0
     8M     16M            218             88          130            0
    16M     32M            148             74           74            0
    32M     64M              4              4            0            0
-----------------------------------------------------------------------
```

从柱状图中可以看出，一些很小的排序也使用了硬盘排序，看来 PGA_AGGREGATE_TARGET 参数确实有些偏小了。那么如何调整这个参数呢？接下来，查看 STATSPACK 报告中的 PGA 建议，如图 7-3 所示。

PGA Target Est (MB)	Size Factr	W/A MB Processed	Estd Extra W/A MB Read/ Written to Disk	Estd PGA Cache Hit %	Estd PGA Overalloc Count
1,280	0.1	113,513,637.6	5,198,930.2	96.0	350,462
2,560	0.3	113,513,637.6	1,154,187.9	99.0	93,607
5,120	0.5	113,513,637.6	832,125.5	99.0	63,379
7,680	0.8	113,513,637.6	552,639.5	99.0	37,120
10,240	1.0	113,513,637.6	88,382.3	99.0	7,583
12,288	1.2	113,513,637.6	0.0	100.0	0
14,336	1.4	113,513,637.6	0.0	100.0	0
16,384	1.6	113,513,637.6	0.0	100.0	0
18,432	1.8	113,513,637.6	0.0	100.0	0
20,480	2.0	113,513,637.6	0.0	100.0	0
30,720	3.0	113,513,637.6	0.0	100.0	0
40,960	4.0	113,513,637.6	0.0	100.0	0
61,440	6.0	113,513,637.6	0.0	100.0	0
81,920	8.0	113,513,637.6	0.0	100.0	0

图　7-3

不难看出，当 PGA_AGGREGATE_TARGET 参数值为 12288 MB 时，PGA 的命中率可以达到 100%，而且 OVERALLOC COUNT 也为 0，因此，我们可以将 PGA_AGGREGATE_TARGET 加大为 12288 MB，如果物理内存充足，在调整 PGA 的时候，可以将这个参数设置得略高一些，比如设置为 14000 MB。

有些时候，PGA 的总体设置没有问题，不过偶尔会出现某个 SQL 存在较为严重的硬盘排序，这该如何分析呢？假设我们知道这个 SQL 的 SQL_ID，那么就可以通过代码清单 7-1 所示的 SQL 进行查询。

代码清单 7-1

```
col op format a15 trunc
col policy format a8 trunc
col last format a10 trunc
set numwidth 8
set line 200
select operation_type as op, operation_id as id, policy,
round(estimated_optimal_size/1024/1024,2) as e_opt,
round(estimated_onepass_size/1024/1024,2) as e_one,
round(last_memory_used/1024/1024,2) as l_mem, last_execution as last,
total_executions as tot, optimal_executions as opt, onepass_executions as one,
multipasses_executions as mult,
round(active_time/1000000,2) as sec, round(max_tempseg_size/1024/1024,2) as tmp_m,
round(last_tempseg_size/1024/1024,2) as tmp_L
from v$sql_workarea where sql_id='37qjh5yuha3x9';
```

具体示例如下：

```
OP                E_OPT  E_ONE  L_MEM  LAST      TOT  OPT  ONE  MULT   SEC   TMP_M TMP_L
---------------- ----- ------- ------ ------- ---- ---- ----- ----- ------ ----- -----
SORT (v2)         59.92  2.63  53.26  OPTIMAL   1    1    0     0   289.53
GROUP BY (HASH)  136.51  8.69  20.1   1 PASS    1    0    1     0   382.3   128   128
```

我们可以从查询结果中看到，这个 SQL 进行了一次排序操作，一次 group by（HASH）操作，其中排序操作是优化模式的，使用了约 53 MB 的内存，group by 操作是 One-Pass 模式的，使用了 20 MB 的内存和 128 MB 的临时段。

如果需要查看目前是哪些 SQL 使用了较多的临时段，我们可以使用如代码清单 7-2 所示的脚本完成进一步分析。

代码清单 7-2

```
select sql_id,operation_type as op, operation_id as id,
  round(estimated_optimal_size/1024/1024,2) as e_opt,
  round(estimated_onepass_size/1024/1024,2) as e_one,
  round(last_memory_used/1024/1024,2) as l_mem,
  Last_execution as last,
  total_executions as tot, optimal_executions as opt,
  onepass_executions as one,
  multipasses_executions as mult,
  round(active_time/1000000,2) as sec,
```

```
  round(max_tempseg_size/1024/1024,2) as tmp_m,
  round(last_tempseg_size/1024/1024,2) as tmp_L
from v$sql_workarea
where
  max_tempseg_size is not null
order by max_tempseg_size desc;
  SQL_ID        OP           ID   E_OPT  E_ONE   L_MEM   LAST  TOT OPT O  M    SEC    TMP_M TMP_L
----------- ------------ --- ------ ----- ------ ------ -- -- - - ------- ---- ----
c940m2fhfdhqb HASH-JOIN       2  2048    33.92  1123.07 1 PASS  1  0  1 0  2515.02  3520  3520
9n9h9vbfsutgf GROUP BY(SORT)  1  1147.91 14.09    97.62 1 PASS  1  0  1 0   481.01  1792  1792
33hqyxbdddcj6 GROUP BY(HASH)  1  1130.5  28.91   142.9  1 PASS  1  0  1 0  2957.31  1088  1088
czmtvcbdamr8v HASH-JOIN       4  1412.22 34.26  1251.05 1 PASS  1  0  1 0  8726.81   768   768
71mcs1y2pguza HASH-JOIN       7   975.08 19.88   432.83 1 PASS  1  0  1 0  3071.57   768   768
47m5kq4hw5f79 GROUP BY(SORT) 13   324.85  7.01    97.62 1 PASS  1  0  1 0   376.89   448   448
gk97kydxcdhf9 GROUP BY(HASH)  2   726.91 22.47   126.81 1 PASS  1  0  1 0   206.21   448   448
7n1hu6k66a369 HASH-JOIN      13   967.37 16.18  1108.92 1 PASS  1  0  1 0   196.07   448   448
gs4aa3r85upzc GROUP BY(HASH)  1   352.85 14.63    88.09 1 PASS  1  0  1 0   270.71   384   384
cmfzdunmgr1a1 GROUP BY(HASH)  1   278.17 14.23    36.15 1 PASS  1  0  1 0    39.38   320   320
3j6f35gyq3syd GROUP BY(HASH)  1   321.92 15.43    42.42 1 PASS  1  0  1 0   288.28   320   320
0tzdnch3vc8tv WINDOW(SORT)    2   355.88  6.1     97.62 1 PASS  2  0  2 0 29026.58   320   320
0tzdnch3vc8tv WINDOW(SORT)    2   323.88  5.83    97.62 1 PASS  1  0  1 0  4853.2    320   320
```

第 8 章

理解 ASM 的结构

ASM 技术的出现曾经给 DBA 带来了巨大的挑战。由于 ASM 像黑匣子一样，我们并不知道里面装的是什么，因此一旦出现问题，我们就束手无策了。早期使用 ASM 技术的用户无一例外地都有过深刻的教训。因此，在 ASM 刚刚出现的几年里，老白经常劝诫客户尽可能不要使用它。随着这些年我们对 ASM 认识的加深，逐渐掌握了其内在结构和原理，也正是这样，我们才可以很轻松地面对客户系统中出现的 ASM 问题，使用 kfed 和 amdu 等工具为客户解决问题，拯救数据。在这个过程中，我们也依靠自己掌握的知识完成了一些项目。

仔细算一算，ASM 出现已经差不多 10 年了，而且 Oracle 也会在不远的将来彻底抛弃裸设备，ASM 很可能会成为大多数系统的首选。掌握 ASM 内部原理对于维护 ASM 磁盘组至关重要，因此老白希望能够把这些知识传授给大家，于是就邀请了老储（储学荣）来编写本章。老储在开发数据扫描工具时曾经认真分析过 ASM 的结构，因此由他来编写这一章，是最为合适的。希望每个维护 ASM 磁盘组的 DBA 都能够认真阅读下面的内容，并自己尝试进行一些测试，本章所介绍的内容将为你提供很大的帮助。

8.1 什么是 ASM

以前的 DBA 一般都有这样的经验：将表和索引分别建在不同的表空间对改善 I/O 性能是有好处的。但如何规划表空间及其包含的数据文件却让人头疼不已。系统上线后经常发现磁盘访问热度不一，有的很忙碌，经常高居 100%，而大多数磁盘却很空闲，I/O 瓶颈很难得到彻底解决。为了帮助用户解决这一问题，Oracle 10*g* 推出了全新的数据库存储技术——ASM（自动存储管理）。它主要用于解决数据库文件的配置、管理和性能难题。我们知道，Oracle 数据库是由一系列文件构成的，主要包括参数文件、控制文件、日志文件、数据文件、归档日志等。在规划数据库时，数据库文件的设计对性能的影响很大，因此需要数据库专家绞尽脑汁，考虑文件采用何种形式和如何分布，但效果往往不佳。而 ASM 的出现，大大降低了数据库规划的复杂性，而且能取得较好的效果。因此，认为 ASM 技术是 Oracle 发展历史上的一个里程碑，并不为过。

我们知道，在 ASM 出现以前，数据库文件从形式上可划分为普通文件和裸设备。普通文件是存在于文件系统中的文件，当然，文件系统可以有多种，比如 AIX 中的 JFS2、LINUX 中的 EXT3，WINDOWS 中的 NTFS，还有网络文件系统 NFS 等。它们尽管各有差异，但是访问接口类似，都

是以目录树的形式组织的，可以通过简单的命令实现创建、移动或删除操作，使用起来较为简单，但因为增加了额外的文件系统开销，所以性能一般。而裸设备是指，操作系统上的一个磁盘设备或逻辑卷以字符设备的形式存在并提供直接访问。由于对裸设备的访问是直接的，不需要任何额外的开销，因此其性能比文件的访问好一些，大概可以提升 8%左右，当然这只是一些内部测试数据，并不总是成立。裸设备的不足之处也是显而易见的，就是难于管理，而且很不灵活，需要事先创建好各个裸设备，如果在一台服务器上存在多个数据库，还容易误用。鉴于 10*g* 版本以前的数据库文件管理和使用存在这样或那样的缺陷，Oracle 在推出 10*g* 版本时带来了革命性的技术——ASM。首先，ASM 解决了裸设备难于使用的问题，我们只需要将用于数据库文件的磁盘设备或逻辑卷放入磁盘组（disk group）中即可，在需要创建数据库文件时，只需指定文件所在的磁盘组，ASM 将自动在磁盘组中创建数据库文件，这大大简化了数据库文件的管理。其次，ASM 较好地解决了数据库文件访问的性能问题，这得益于 ASM 的 SAME 特性，SAME 是 Stripe And Mirror Everything 的缩写，即条带化并镜像一切。也就是说，在磁盘组中创建数据库文件时，ASM 会对文件进行条带化处理，按照文件的类型，选取合适的条带值，将文件分布在所有的磁盘上，这样就很好地避免了因某些磁盘访问过于频繁而造成 I/O 性能下降的问题。最后，ASM 提供了更好的容错性，避免单点失败导致数据丢失的情况出现。这是因为 ASM 在磁盘组中设置了不同的失败组，每个文件根据文件类型和冗余度的设置，其内容可以在不同的失败组中存在一个或多个副本（mirror），这样，即使某个磁盘出现问题导致文件的部分内容不能被正常访问，ASM 也能自动从其他失败组上访问该文件的镜像内容。这里以表格的形式对三者进行比较，具体内容如表 8-1 所示。

表　8-1

	文件系统	裸　设　备	ASM
易用性	好	差	好
性能	一般	较好	好
容错性	差	差	好
RAC支持	差	一般	好

另外，ASM 还具有强大的存储能力，可以达到：

- 63 磁盘组；
- 10 000 磁盘；
- 4PB/磁盘；
- 40EB 总容量；
- 1 000 000 文件/磁盘组；
- 2.4TB/文件，而 11*g* 版本更扩展到了 128TB。

综上所述，ASM 是目前最好的 Oracle 数据库存储技术。当然，和任何新生技术一样，ASM 刚推出时，由于存在较多的 Bug，有时会导致元数据出错而造成数据库不能访问，严重时甚至造成数据库损坏。另外，由于 Oracle 提供的 asmcmd 工具功能有限，和外部的文件系统进行文件交换很不方便，因此，Oracle 10*g* 的用户对 ASM 的接受程度比较低，都不想当“先驱”。但随着技

术的不断完善，到 11*g* 版本时，ASM 技术已比较成熟，不但变得更为健壮，而且 asmcmd 工具的功能有了大幅提升，还提供了和文件系统类似的命令语法。

ASM 和 ASMLIB

ASM 的存储结构如图 8-1 所示。

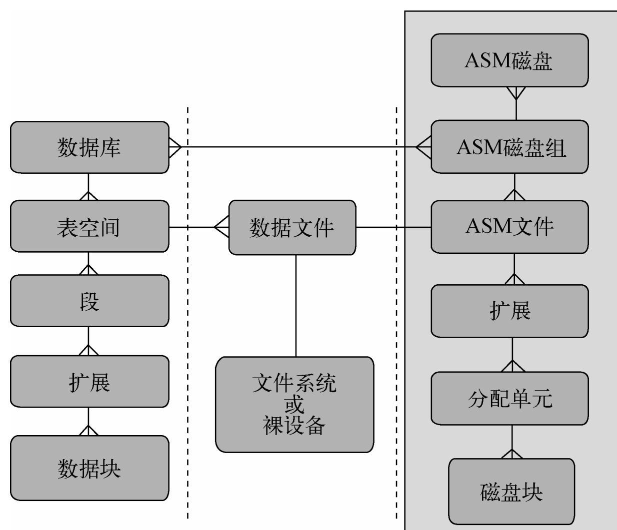


图　8-1

ASM 的管理单位是磁盘组，类似操作系统的卷组，每个磁盘组由多个磁盘构成，类似构成卷组的物理卷，ASM 磁盘可以使用以下存储资源：

- 整个磁盘或磁盘分区；
- 存储阵列上划分的逻辑单元（LUN）；
- 逻辑卷；
- 网络文件系统（NFS）。

这些磁盘可以组织成多个失败组，顾名思义，失败组的主要目的是为了容错，因为 ASM 的 SAME 特性，ASM 文件的扩展可以存在一个或多个镜像，每个镜像放在不同的失败组，这样即使某个失败组上的磁盘无法访问，ASM 也可以通过访问位于其他失败组的镜像加以恢复。磁盘组在逻辑上组织成一个大的文件系统，以 ASM 文件的方式管理上面的内容，ASM 文件类似普通的操作系统文件，也有目录名和 I-NODE（主要保存扩展分配表和文件属性）。

ASM 文件由扩展组成，扩展是一组连续的分配单元，扩展的大小一般为 1 个分配单位，但是对于大型文件，就需要很大的分配表，这些分配表在文件打开时被放在数据库的共享池中，会消耗很多内存，因此，为了支持 VLDB，ASM 采用了多种大小的扩展，对于超过 20 000 个扩展的文件，超出部分将采用 4 个分配单位的扩展，对于超过 40 000 个扩展的文件，超出部分将采用

16 个分配单位的扩展。

我们知道，ASM 具有 SAME 特性，而 ASM 文件镜像的单位是扩展，在分配表中，记录了每个镜像所处的扩展。也就是说，假如某个 ASM 文件存在一个镜像的话，在形式上并不像数据库日志文件的镜像那样存在两个文件，还是只有一个文件，只不过文件的分配表中记录了两份保存同样内容的扩展，这些扩展位于不同的失败组。分配单元是最小的 I/O 单位，类似于数据库的块，默认是 1 MB，最大可设置到 64 MB。ASM 在进行条带化时，一般也是以分配单元为单位进行分割的。每个分配单元在物理上由一组连续的磁盘块构成。

对于使用 ASM 存储数据的数据库而言，数据库的数据文件、重做日志、归档日志等文件都能以 ASM 文件的形式存储在 ASM 磁盘组中，并且可以存放在不同的磁盘组中。而到了 Oracle 11*g* 版本，连 OCR 和 Voting 文件都可以保存在磁盘组中。

像数据库一样，ASM 也是以实例的形式存在，对 ASM 存储进行管理，并向 ASM 的使用者提供服务。ASM 实例结构如图 8-2 所示。

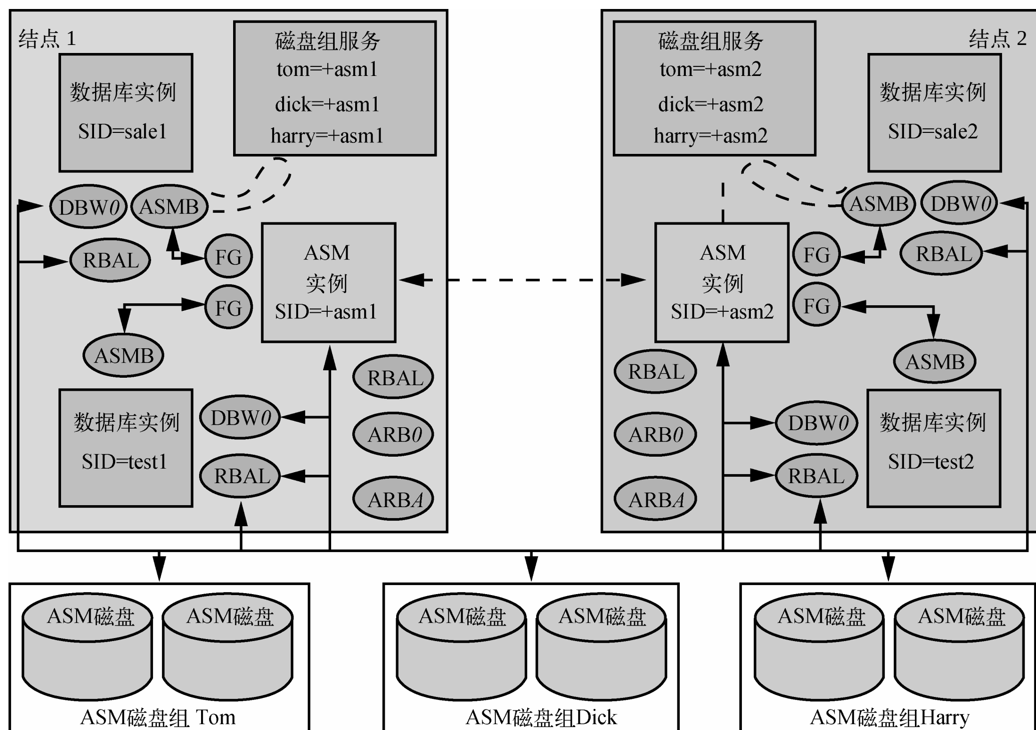


图　8-2

上图列出了 ASM 实例的一些特有的后台进程，至于一般的后台进程，比如 SMON、PMON 等，因为其功能和数据库实例的那些后台进程类似，所以这里没有一一列出。在 ASM 实例中，存在以下几种新的后台进程。

- RBAL：重平衡进程，当有磁盘加入或退出时，对 ASM 文件的扩展重新分布，以达到平衡磁盘访问的效果。
- ARB*x*：平衡子进程，具体实施平衡操作的一组子进程。

在数据库实例中，存在以下几种新的后台进程。

- ASMB：从 CSS 处获取 ASM 实例连接串，然后创建到 ASM 实例的连接，和 ASM 进行交互，比如更新统计信息，获取文件扩展分配表等。
- RBAL：全局性地打开 ASM 磁盘组中的所有磁盘，供后面的操作使用。
- O0*nn*：一组连接到 ASM 实例的子进程，构成连接池，加快和 ASM 实例的交互。

需要注意的是，ASM 只是向数据库提供了 ASM 文件的空间分配和文件管理服务，并不接管文件的读写功能，因此，具体的文件读写还是由数据库进程执行的。比如，数据脏块的写操作由 dbwr 进程完成，日志记录的写操作由 lgwr 进程完成，归档日志的写操作由 archive 或服务器进程完成。

ASMLIB 是 Oracle 公司提出的一个供 ASM 使用的 API 规范和接口库，它主要包括磁盘管理和识别、磁盘访问、性能和可靠性优化等功能。如果没有 ASMLIB，ASM 需要使用操作系统底层的系统调用访问磁盘，这样，对于不同的操作系统和存储，存在着不一致的访问方式，并且不能充分发挥存储硬件的能力。因此，Oracle 推出了一套规范和接口，命名为 ASMLIB，向所有操作系统和存储厂家开放，希望他们能提供具体的实现库，但好像响应的并不多。目前，除了 Oracle 自己在 Linux 平台推出的参考实现外，我们还没有安装过其他厂家实现的 ASMLIB。

8.2　ASM 的结构

ASM 的 METADATA 是由一系列内部文件组成的，其中 FILE 0 就是 ASM DISKHEADER。本节将从 FILE 0 开始，逐步介绍 ASM 的各类文件的数据结构，通过本节内容的学习，有兴趣的朋友可以编写一个自己的 ASM 数据分析和数据抽取工具，以便在 ASM 故障时使用这个工具进行数据抽取。当然也可以直接使用 Oracle 的 AMDU 工具，不过有些时候，AMDU 对数据一致性的要求太高，如果 AMDU 拒绝为你“干活”，那么就只能使用自己编写的工具了。

8.2.1　ASM DISKHEADER 的结构

ASM 通过一系列元数据进行管理，这些元数据类似于数据库的数据字典。元数据从形式上分为物理元数据和虚拟元数据。所谓物理元数据，是指记录在磁盘固定位置的数据，主要包括磁盘头、分配表、空闲空间表和伙伴状态表。而虚拟元数据是指以文件的形式存在的元数据，其位置不固定，主要包括文件目录、磁盘目录、活动变化目录、继续操作目录、模板目录、别名目录、属性目录、过期目录、注册表等。元数据的每个块为 4096 B，每个块都有相应的结构，这些结构随版本的升级会略有扩展，以下介绍的数据结构是基于 Oracle 10*g* R2 版本的。在所有的元数据中，磁盘头信息都是元数据的起点，通过它的引导，可以一步步地遍历所有的元数据。磁盘头的数据结构由两部分组成，第一部分是块头，共 32 B，块头结构 kfbh 如下：

```
typedef struct kfbh_ {
  ub1                  endian_kfbh;
  ub1                  hard_kfbh;
  kfbtyp               type_kfbh;
  ub1                  datfmt_kfbh;
  kfbl                 block_kfbh;
  ub4                  check_kfbh;
  kfcn                 fcn_kfbh;
  ub4                  spare1_kfbh;
  ub4                  spare2_kfbh;
} kfbh;
```

kfbh 中的一些主要的字段如下所示。

- endian_kfbh：平台软件的字节顺序（大小端）。
- hard_kfbh：用于标识的 MAGIC 数。
- type_kfbh：元数据块类型，主要包括 1 种磁盘头、4 种文件目录、6 种磁盘目录、11 种别名目录和 12 种间接扩展。
- datfmt_kfbh：元数据块数据格式。
- block_kfbh：当前块的位置信息，包括所属对象及块号。
- check_kfbh：用于块一致性检查的校验和。
- fcn_kfbh：最后一次改变的改变号，类似于 SCN。

块头是最基本的，每个元数据块都会包含块头。磁盘头的第二部分结构 kfdhdb 包含了主要的磁盘信息，定义如下：

```
typedef struct kfdhdb_ {
  kfddrb               driver_kfdhdb;
  ub4                  compat_kfdhdb;
  ub2                  dsknum_kfdhdb;
  kfdgtp               grptyp_kfdhdb;
  kfdhdr               hdrsts_kfdhdb;
  oratext              dskname_kfdhdb[32];
  oratext              grpname_kfdhdb[32];
  oratext              fgname_kfdhdb[32];
  oratext              capname_kfdhdb[32];
  kfts                 crestmp_kfdhdb;
  kfts                 mntstmp_kfdhdb;
  ub2                  secsize_kfdhdb;
  ub2                  blksize_kfdhdb;
  ub4                  ausize_kfdhdb;
  ub4                  mfact_kfdhdb;
  ub4                  dsksize_kfdhdb;
  ub4                  pmcnt_kfdhdb;
  ub4                  fstlocn_kfdhdb;
  ub4                  altlocn_kfdhdb;
  ub4                  f1b1locn_kfdhdb;
  ub2                  redomirrors_kfdhdb[4];
  ub4                  dbcompat_kfdhdb;
  kfts                 grpstmp_kfdhdb;
  ub4                  ub4spare_kfdhdb[58];
  kfracdb              acdb_kfdhdb;
} kfdhdb;
```

其中一些主要的字段如下所示。

- driver_kfdhdb：驱动程序保留信息。
- compat_kfdhdb：打开本磁盘组所需的 ASM 最小版本。
- dsknum_kfdhdb：磁盘号。
- grptyp_kfdhdb：磁盘组冗余类型，包括外部冗余、正常冗余和高度冗余三种类型。
- hdrsts_kfdhdb：磁盘头状态。
- dskname_kfdhdb：磁盘名。
- grpname_kfdhdb：磁盘组名。
- fgname_kfdhdb：失败组名。
- crestmp_kfdhdb：创建时间戳。
- mntstmp_kfdhdb：Mount 时间戳。
- secsize_kfdhdb：磁盘扇区大小。
- blksize_kfdhdb：元数据块大小。
- ausize_kfdhdb：分配单元大小。
- dsksize_kfdhdb：磁盘大小，以分配单元为单位。
- pmcnt_kfdhdb：物理元数据所占用的分配单元数。
- fstlocn_kfdhdb：空闲空间表的第一个块号。
- altlocn_kfdhdb：分配表的第一个块号。
- f1b1locn_kfdhdb：文件目录指针，文件目录是一个特殊的虚拟元数据文件，它包含了所有 ASM 文件的 I-NODE，当然，它也包含自身的 I-NODE，其文件号是 1。如果本磁盘包含了文件目录的第一个扩展，那么就用该字段指向其所在的第一个分配单元号。
- dbcompat_kfdhdb：打开磁盘组所需的数据库实例最小版本，在数据库实例初始化参数 compatible 中设置了数据库实例的兼容性版本，该版本小于或等于数据库软件的版本，比如 11*g* 版本的数据库实例通过设置 compatible 参数可以运行在 10.2 的兼容性版本下。
- grpstmp_kfdhdb：磁盘组创建时间戳。

8.2.2 ASM FILE DIRECTORY 文件结构

文件目录包含了所有 ASM 文件的 I-NODE 信息，其中最主要的是文件大小和扩展分配表，其文件号是 1。文件目录在结构上是按块组织的，每个块的数据结构由三部分组成。第一部分是块头 kfbh，结构和磁盘头的块头一样，第二部分结构 kfffdb，主要包含了文件分配信息，定义如下：

```
typedef struct kfffdb_ {
  kffbnd              node_kfffdb;
  ub4                 hibytes_kfffdb;
  ub4                 lobytes_kfffdb;
  ub4                 xtntcnt_kfffdb;
  ub4                 xtnteof_kfffdb;
```

```
    ub4                  blksize_kfffdb;
    kfffdf               flags_kfffdb;
    ub1                  filetype_kfffdb;
    kfrs                 dxrs_kfffdb;
    kfrs                 ixrs_kfffdb;
    ub4                  dxsiz_kfffdb[3];
    ub4                  ixsiz_kfffdb[3];
    ub2                  xtntblk_kfffdb;
    ub2                  break_kfffdb;
    ub1                  prizn_kfffdb;
    ub1                  seczn_kfffdb;
    ub2                  ub2spare_kfffdb;
    ub4                  alias_kfffdb[2];
    ub1                  strpwdth_kfffdb;
    ub1                  strpsz_kfffdb;
    ub2                  usmsz_kfffdb;
    kfts                 crets_kfffdb;
    kfts                 modts_kfffdb;
    ub4                  spare_kfffdb[16];
    oratext              usm_kfffdb[1024];
} kfffdb;
```

其中一些主要的字段如下所示。

- node_kfffdb：块的分配信息，包括块的分支号和指向 freelist 中下一个块的指针，分支号是一个内部序列号，每次块被重用都会增加，其作用类似于版本号。
- hibytes_kfffdb：文件字节数（高位）。
- lobytes_kfffdb：文件字节数（低位）。
- xtntcnt_kfffdb：直接扩展数。
- xtnteof_kfffdb：文件 EOF 之前的扩展数。
- blksize_kfffdb：每个块的字节数，不同的文件类型块大小可能不同，比如，元数据文件为 4096 B，数据库日志文件为 512 B，而数据文件一般是 8192 B。
- flags_kfffdb：文件标志。
- filetype_kfffdb：文件类型，主要包括以下几种。
  - 15：元数据文件；
  - 12：数据文件；
  - 13：参数文件；
  - 3：日志文件；
  - 6：临时文件；
  - 1：控制文件。
- dxrs_kfffdb：直接扩展冗余模式，如果采用了正常冗余，每个扩展还会增加一个镜像，而如果采用高度冗余，每个扩展会增加两个镜像。
- ixrs_kfffdb：间接扩展冗余模式。
- dxsiz_kfffdb：每种扩展大小包含的直接扩展数，扩展大小分为 1 个分配单元、4 个分配单元、16 个分配单元三种。

- ixsiz_kfffdb：每种扩展大小包含的间接扩展数。
- xtntblk_kfffdb：本块包含的直接和间接扩展指针数。
- break_kfffdb：直接和间接扩展指针的边界插槽，一般是 60，表示前 60 个是直接扩展指针。
- prizn_kfffdb：主扩展所处磁盘区。
- seczn_kfffdb：镜像扩展所处磁盘区。
- alias_kfffdb：文件别名指针。
- strpwdth_kfffdb：条带宽度，即条带跨越几个扩展，可以简单理解为在几个磁盘间做条带。
- strpsz_kfffdb：条带大小（2^N），比如，20 表示 1 MB 的条带，17 表示 128 KB 的条带。

条带宽度和条带大小决定了条带划分模式，效果如图 8-3 所示。

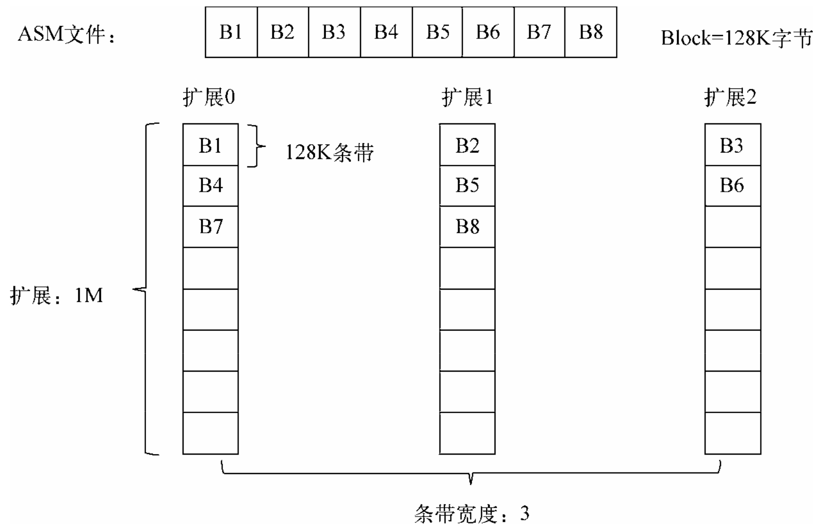


图　8-3

从上图可以看出，1 MB 的文件在做细粒度的条带（128 KB）时需要 3 个扩展，如果做粗粒度的条带（1 MB），则只需要 1 个扩展。

- usmsz_kfffdb：文件附加的用户元数据大小。
- crets_kfffdb：文件创建时间。
- modts_kfffdb：文件修改时间。
- usm_kfffdb：文件附加的用户元数据。

文件目录块的第三部分是一个数组结构 kfxp[360]，最多可以包含 360 个扩展指针，定义如下：

```
typedef struct kfxp_ {
  ub4                 au_kfxp;
  ub2                 disk_kfxp;
  kfxpf               flags_kfxp;
  ub1                 chk_kfxp;
}kfxp;
```

其中一些主要的字段如下所示。

- au_kfxp：扩展的第一个分配单元号。
- disk_kfxp：所属磁盘号。
- flags_kfxp：标志。
- chk_kfxp：用于一致性检查的校验和。

在扩展分配表中，扩展分为直接扩展和间接扩展，直接扩展中存放的是实际的文件内容，而间接扩展中存放的是第二层的扩展分配表，通过引入间接扩展，可以更灵活地表达超大型文件的存放位置。扩展分配表结构如图 8-4 所示。

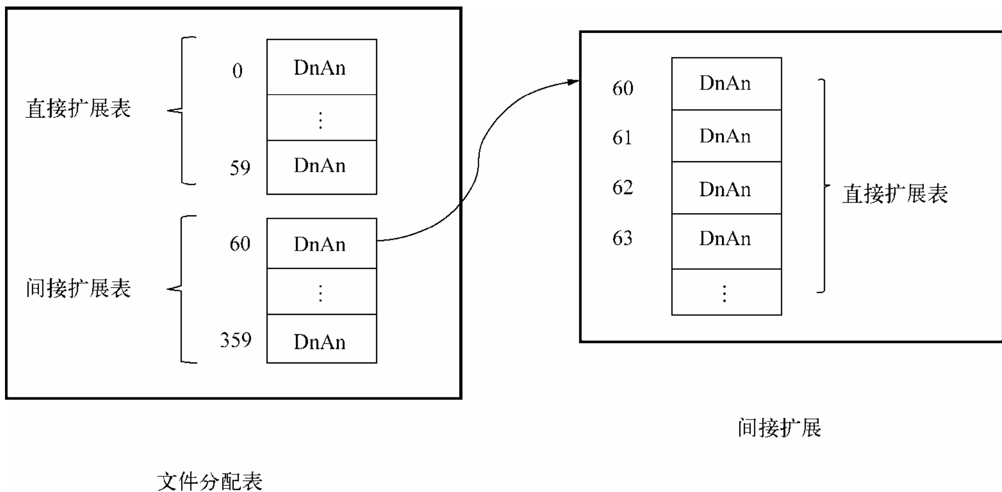


图　8-4

对于间接扩展，每一个扩展块的结构也由三部分组成。第一部分是块头 kfbh，第二部分结构 kffixb 主要包含了文件分配信息，定义如下：

```
typedef struct kffixb_ {
  ub4                   dxsn_kffixb;
  ub2                   xtntblk_kffixb;
  kfrs                  dXrs_kffixb;
  ub1                   ub1spare_kffixb;
  ub4                   ub4spare_kffixb;
} kffixb;
```

其中一些主要的字段如下所示。

- dxsn_kffixb：第一个扩展号。
- xtntblk_kffixb：块中的扩展指针数。
- dXrs_kffixb：扩展冗余模式。

第三部分是一个数组结构 kfxp[480]，最多可以包含 480 个直接扩展指针。

8.2.3　ASM ALIAS DIRECTORY 文件结构

别名目录记录磁盘组中的别名信息，别名可以对应具体的文件，也可以是某个中间路径，其作用类似于文件系统的目录，并且也采用了层次性的形式来表示，其文件号是 6。别名目录在结构上是按块组织的，每个块的数据结构由三部分组成，第一部分是块头 kfbh，其结构和磁盘头的块头一样；第二部分是 kffdnd，主要用来定位，它描述了块在目录树中的位置；第三部分结构是一个数组，包含了在目录树中处于同一层的别名信息，每个目录项（别名）的结构为 kfade。在描述具体的数据结构前，我们可以先用图 8-5 表示目录树的整体结构。

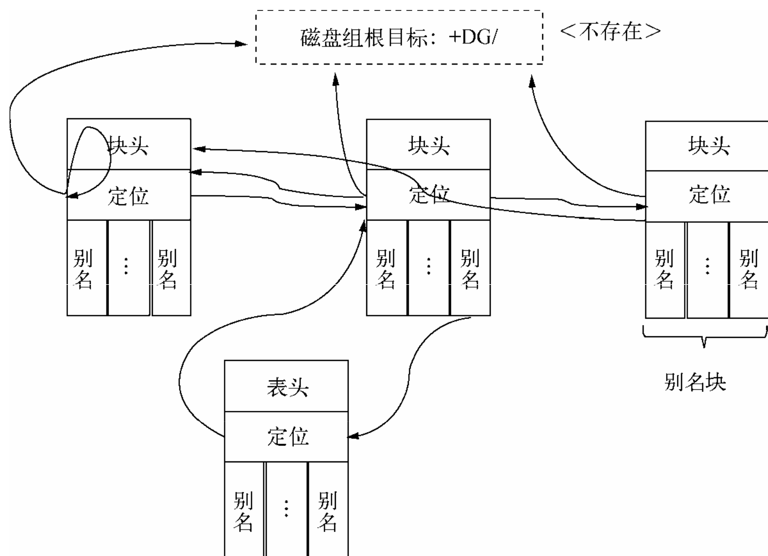


图　8-5

接下来，我们具体看一下用于定位的 kffdnd：

```
typedef struct kffdnd_ {
  kffbnd              bnode_kffdnd;
  kffban              overfl_kffdnd;
  kffban              parent_kffdnd;
  kffban              fstblk_kffdnd;
} kffdnd;
```

其中一些主要的字段如下所示。

- bnode_kffdnd：块的分配信息，包括块的分支号和指向 freelist 中下一个块的指针。
- overfl_kffdnd：指向同一层次的下一个块，即兄弟块。
- parent_kffdnd：指向上一层的块，即父块。
- fstblk_kffdnd：指向同一层次的第一个块。

别名 kfade 的定义如下：

```
typedef struct kfade_ {
  kffden              entry_kfade;
  oratext             name_kfade[48];
  ub4                 fnum_kfade;
  ub4                 finc_kfade;
  kfadef              flags_kfade;
  ub1                 ub1spare_kfade;
  ub2                 ub2spare_kfade;
} kfade;
```

其中一些主要的字段如下所示。

- entry_kfade：每个表项的通用头结构，包括分支号、HASH 值和指向下一层块的指针，即子块。
- name_kfade：名称。
- fnum_kfade：对应文件号，如果别名对应的是目录，该字段无用。
- finc_kfade：文件分支号，文件号被复用时，会自动增加分支号以示区别。
- flags_kfade：标志，可以区分别名对应的是文件还是目录。

此处以文件+DG/ora/init/spfile 为例，来分析 ASM 怎样在别名目录中找到相应的别名。

(1) +DG 根路径表示文件位于 DG 磁盘组。

(2) 从别名目录的第一个块开始，扫描其中的别名数组寻找 ora 别名，如果未找到，根据块的 kffdnd 结构中的 overfl_kffdnd 指针找到兄弟块，接着扫描，直到找到为止。

(3) 根据 ora 别名结构中的 entry_kfade 指针找到下一层路径的块，寻找 init 别名。

(4) 根据 init 别名结构中的 entry_kfade 指针找到下一层路径的块，找到 spfile 别名，至此，搜寻结束。

在 ASM 中，别名默认由系统自动生成，根据文件类型，系统在合适的目录层次中为文件产生别名，其形式为：

+<磁盘组>/<数据库名>/<分类>/<标记>.<文件号>.<文件分支>

具体的产生规则如表 8-2 所示。

表 8-2

Oracle文件类型	分 类	标 记	默认模板
控制文件	controlfile	CF/BCF	CONTROLFILE
数据文件	datafile	<表空间名>_<文件号>	DATAFILE
在线日志	online_log	log_<线索号>	ONLINELOG
归档日志	archive_log	parameter	ARCHIVELOG
临时文件	temp	<表空间名>_<文件号>	TEMPFILE
数据文件备份分片	backupset	客户端指定	BACKUPSET
数据文件增量备份分片	backupset	客户端指定	BACKUPSET
归档日志备份分片	backupset	客户端指定	BACKUPSET
数据文件拷贝	datafile	<表空间名>_<文件号>	DATAFILE
初始化参数文件	init	Spfile	PARAMETERFILE

（续）

Oracle文件类型	分　类	标　记	默认模板
DG Broker配置文件	Drc	drc	DATAGUARDCONFIG
闪回日志文件	rlog	<线索号>_<日志号>	FLASHBACK
备份后块改变跟踪位图	CTB	BITMAP	CHANGETRACKING
控制文件自动备份	AutoBackup	客户端指定	AUTOBACKUP
数据泵导出文件	Dumpset	Dump	DUMPSET
跨平台转换数据文件			XTRANSPORT
OCR文件	ocrfile		OCRFILE

当然，别名也可以在文件创建时显式指定别名，这时需要提供完整的路径名，从磁盘组开始，中间的目录必须是已经存在的，如果不存在的话，要先手工创建这些目录。

8.2.4　ASM DISK DIRECTORY 文件结构

磁盘目录记录磁盘组中所有磁盘的信息，其文件号是2。磁盘目录在结构上是按块组织的，和别名目录的组织形式相同，每个块的数据结构也由三部分组成，第一部分是块头 kfbh，其结构和别名目录的块头是一样的；第二部分的结构和别名目录的 kffdnd 结构也是一样的；第三部分是一个数组，数组的每一项都包含了一个磁盘的信息，其结构 kfddde 定义如下：

```
typedef struct kfddde_ {
  kffden                entry_kfddde;
  ub2                   dsknum_kfddde;
  kfdsta                state_kfddde;
  ub1                   ub1spare_kfddde;
  oratext               dskname_kfddde[32];
  oratext               fgname_kfddde[32];
  kfts                  crestmp_kfddde;
  kfts                  failstmp_kfddde;
  ub4                   timer_kfddde;
  ub4                   size_kfddde;
  ub4                   spare_kfddde[5];
  kfdzon                zones_kfddde[4];
} kfddde;
```

其中一些主要的字段如下所示。

- entry_kfddde：每个表项的通用头结构，包括分支号、散列值、指向下一层块的指针。
- dsknum_kfddde：磁盘号。
- state_kfddde：磁盘状态。
- dskname_kfddde：磁盘名。
- fgname_kfddde：失败组名。
- crestmp_kfddde：创建时间戳。
- failstmp_kfddde：失败时间戳。

- timer_kfddde：失败定时器。
- size_kfddde：分配单元数。
- zones_kfddde：整个磁盘按照冷热程度被划分为四个区，该数组存放每个区的布局及使用情况。

8.2.5　从 ASM 存储结构谈 ASM 日常维护的要点

通过前面几节对 ASM 存储结构的介绍，我们可以总结出以下一些有用的经验。

同一磁盘组采用性能接近的磁盘。鉴于 ASM 条带化并且镜像一切的特性，每个文件都会分布在所有磁盘上，因此如果某个硬盘性能较差，就会形成瓶颈，进而影响整体的 I/O 性能。

- 同一磁盘组采用大小接近的磁盘。ASM 条带化并且镜像一切的特性，会把文件尽量平均分布在所有磁盘上，因此如果某个硬盘容量偏小，就有可能发生 ORA-15041 错误（磁盘组空间耗尽），尽管这时其他硬盘可能还有空闲空间。
- 失败组的容量应该相同，否则 ASM 将只会使用最小的那个失败组的大小，其他空间将不会被使用，从而造成容量浪费。
- 不同的失败组在物理上应相对隔离，比如，置于不同的磁盘阵列，使用不同的 HOST BUS ADAPTER 连接等，防止发生单点故障。
- 如果存储自身未实施硬件 RAID 保护，建议采用 ASM 镜像进行冗余。
- 磁盘头信息不要被操作系统覆盖。有些操作系统会在磁盘头存放一些控制信息，这些信息有可能会覆盖掉 ASM 磁盘头信息，因此在实施 ASM 时，需要了解这些情况，并采取相应措施来规避，比如，可以为磁盘建几个分区，使用后面的分区作为 ASM 磁盘等。
- 将数据库文件和用于数据库恢复的文件（归档日志、RMAN 备份等）放在不同的磁盘组。
- 需要扩充磁盘时，尽量一次加入多个磁盘。每次加入磁盘时，ASM 都会进行重新平衡操作，把其他磁盘上的一部分扩展转移到新加入的磁盘中，这会带来一些开销。因此一次加入多个磁盘意味着减少重新平衡的次数，相应地，也就减少了额外的开销。
- 定期检查 ASM 实例和数据库实例的 ALERT 文件，及时解决出现的磁盘故障。
- 有些操作系统为同一个磁盘设备定义了几个设备名，每个设备名使用不同的路径访问同一个磁盘，这可以增加 I/O 吞吐量。但是 ASM 自身是无法进行多路径识别的，这些不同路径会被 ASM 认为是不同的磁盘，因此需要操作系统为支持多路径的设备提供伪设备名供 ASM 使用。
- 直接使用磁盘或 LUN，不需要使用逻辑卷。
- 关注 Oracle 公布的 ASM 相关的 Bug，必要时安装补丁包。
- 熟悉相关的 ASM 工具，这些工具可能会为你提供很大的帮助，推荐的工具主要包括：
  - ASMCMD 实用命令行工具；
  - AMDU 数据拯救工具；
  - KFED 元数据编辑工具。

8.3 如何使用 KFED 分析和修改 ASM 数据

KFED 是 Oracle 提供的检查、修改 ASM 元数据的实用工具，类似于修改数据块的 bbed 工具。KFED 工具可以通过以下途径获取：

- 对于 UNIX/LINUX 系统，可通过 make 命令生成。

```
cd $ORACLE_HOME/RDBMS/lib
make –f ins_RDBMS.mk ikfed
```

此时在$ORACLE_HOME/bin 目录下将会生成 KFED 可执行程序。

- 对于 Windows 系统，可从 Metalink 下载 kfedwin.exe 可执行程序。

下面简单介绍 KFED 工具的一些常见使用场景。

1. 读取 ASM 元数据信息

以下命令用于读取磁盘头元数据：

```
[grid@localhost ~]$ kfed read /dev/oracleasm/disks/ASMDISK1 aunum=0 blknum=0
kfbh.endian:                          1 ; 0x000: 0x01
kfbh.hard:                          130 ; 0x001: 0x82
kfbh.type:                            1 ; 0x002: KFBTYP_DISKHEAD
kfbh.datfmt:                          1 ; 0x003: 0x01
kfbh.block.blk:                       0 ; 0x004: T=0 NUMB=0x0
kfbh.block.obj:              2147483648 ; 0x008: TYPE=0x8 NUMB=0x0
kfbh.check:                  2779965978 ; 0x00c: 0xa5b2ea1a
kfbh.fcn.base:                     6940 ; 0x010: 0x00001b1c
kfbh.fcn.wrap:                        0 ; 0x014: 0x00000000
kfbh.spare1:                          0 ; 0x018: 0x00000000
kfbh.spare2:                          0 ; 0x01c: 0x00000000
kfdhdb.driver.provstr: ORCLDISKASMDISK1 ; 0x000: length=16
kfdhdb.driver.reserved[0]:   1145918273 ; 0x008: 0x444d5341
kfdhdb.driver.reserved[1]:    827020105 ; 0x00c: 0x314b5349
kfdhdb.driver.reserved[2]:            0 ; 0x010: 0x00000000
kfdhdb.driver.reserved[3]:            0 ; 0x014: 0x00000000
kfdhdb.driver.reserved[4]:            0 ; 0x018: 0x00000000
kfdhdb.driver.reserved[5]:            0 ; 0x01c: 0x00000000
kfdhdb.compat:                186646528 ; 0x020: 0x0b200000
kfdhdb.dsknum:                        0 ; 0x024: 0x0000
kfdhdb.grptyp:                        2 ; 0x026: KFDGTP_NORMAL
kfdhdb.hdrsts:                        3 ; 0x027: KFDHDR_MEMBER
kfdhdb.dskname:               DATA_0000 ; 0x028: length=9
kfdhdb.grpname:                    DATA ; 0x048: length=4
kfdhdb.fgname:                DATA_0000 ; 0x068: length=9
kfdhdb.capname:                         ; 0x088: length=0
kfdhdb.crestmp.hi:         32958036 ; 0x0a8: HOUR=0x14 DAYS=0x12 MNTH=0x9 YEAR=0x7db
kfdhdb.crestmp.lo:       1372083200 ; 0x0ac: USEC=0x0 MSEC=0x215 SECS=0x1c MINS=0x14
kfdhdb.mntstmp.hi:         32968205 ; 0x0b0: HOUR=0xd DAYS=0x10 MNTH=0x3 YEAR=0x7dc
kfdhdb.mntstmp.lo:        438844416 ; 0x0b4: USEC=0x0 MSEC=0x20f SECS=0x22 MINS=0x6
kfdhdb.secsize:                     512 ; 0x0b8: 0x0200
kfdhdb.blksize:                    4096 ; 0x0ba: 0x1000
kfdhdb.ausize:                  1048576 ; 0x0bc: 0x00100000
kfdhdb.mfact:                    113792 ; 0x0c0: 0x0001bc80
```

```
kfdhdb.dsksize:                    7640 ; 0x0c4: 0x00001dd8
kfdhdb.pmcnt:                         2 ; 0x0c8: 0x00000002
kfdhdb.fstlocn:                       1 ; 0x0cc: 0x00000001
kfdhdb.altlocn:                       2 ; 0x0d0: 0x00000002
kfdhdb.f1b1locn:                      2 ; 0x0d4: 0x00000002
kfdhdb.redomirrors[0]:                0 ; 0x0d8: 0x0000
kfdhdb.redomirrors[1]:                1 ; 0x0da: 0x0001
kfdhdb.redomirrors[2]:                2 ; 0x0dc: 0x0002
kfdhdb.redomirrors[3]:            65535 ; 0x0de: 0xffff
kfdhdb.dbcompat:              168820736 ; 0x0e0: 0x0a100000
kfdhdb.grpstmp.hi:         32958036 ; 0x0e4: HOUR=0x14 DAYS=0x12 MNTH=0x9 YEAR=0x7db
kfdhdb.grpstmp.lo:       1371925504 ; 0x0e8: USEC=0x0 MSEC=0x17b SECS=0x1c MINS=0x14
kfdhdb.vfstart:                       0 ; 0x0ec: 0x00000000
kfdhdb.vfend:                         0 ; 0x0f0: 0x00000000
kfdhdb.spfile:                       59 ; 0x0f4: 0x0000003b
kfdhdb.spfflg:                        1 ; 0x0f8: 0x00000001
kfdhdb.ub4spare[0]:                   0 ; 0x0fc: 0x00000000
kfdhdb.ub4spare[1]:                   0 ; 0x100: 0x00000000
kfdhdb.ub4spare[2]:                   0 ; 0x104: 0x00000000
kfdhdb.ub4spare[3]:                   0 ; 0x108: 0x00000000
kfdhdb.ub4spare[4]:                   0 ; 0x10c: 0x00000000
kfdhdb.ub4spare[5]:                   0 ; 0x110: 0x00000000
kfdhdb.ub4spare[6]:                   0 ; 0x114: 0x00000000
kfdhdb.ub4spare[7]:                   0 ; 0x118: 0x00000000
kfdhdb.ub4spare[8]:                   0 ; 0x11c: 0x00000000
kfdhdb.ub4spare[9]:                   0 ; 0x120: 0x00000000
kfdhdb.ub4spare[10]:                  0 ; 0x124: 0x00000000
kfdhdb.ub4spare[11]:                  0 ; 0x128: 0x00000000
kfdhdb.ub4spare[12]:                  0 ; 0x12c: 0x00000000
kfdhdb.ub4spare[13]:                  0 ; 0x130: 0x00000000
kfdhdb.ub4spare[14]:                  0 ; 0x134: 0x00000000
kfdhdb.ub4spare[15]:                  0 ; 0x138: 0x00000000
kfdhdb.ub4spare[16]:                  0 ; 0x13c: 0x00000000
kfdhdb.ub4spare[17]:                  0 ; 0x140: 0x00000000
kfdhdb.ub4spare[18]:                  0 ; 0x144: 0x00000000
kfdhdb.ub4spare[19]:                  0 ; 0x148: 0x00000000
kfdhdb.ub4spare[20]:                  0 ; 0x14c: 0x00000000
kfdhdb.ub4spare[21]:                  0 ; 0x150: 0x00000000
kfdhdb.ub4spare[22]:                  0 ; 0x154: 0x00000000
kfdhdb.ub4spare[23]:                  0 ; 0x158: 0x00000000
kfdhdb.ub4spare[24]:                  0 ; 0x15c: 0x00000000
kfdhdb.ub4spare[25]:                  0 ; 0x160: 0x00000000
kfdhdb.ub4spare[26]:                  0 ; 0x164: 0x00000000
kfdhdb.ub4spare[27]:                  0 ; 0x168: 0x00000000
kfdhdb.ub4spare[28]:                  0 ; 0x16c: 0x00000000
kfdhdb.ub4spare[29]:                  0 ; 0x170: 0x00000000
kfdhdb.ub4spare[30]:                  0 ; 0x174: 0x00000000
kfdhdb.ub4spare[31]:                  0 ; 0x178: 0x00000000
kfdhdb.ub4spare[32]:                  0 ; 0x17c: 0x00000000
kfdhdb.ub4spare[33]:                  0 ; 0x180: 0x00000000
kfdhdb.ub4spare[34]:                  0 ; 0x184: 0x00000000
kfdhdb.ub4spare[35]:                  0 ; 0x188: 0x00000000
kfdhdb.ub4spare[36]:                  0 ; 0x18c: 0x00000000
kfdhdb.ub4spare[37]:                  0 ; 0x190: 0x00000000
```

```
kfdhdb.ub4spare[38]:                  0 ; 0x194: 0x00000000
kfdhdb.ub4spare[39]:                  0 ; 0x198: 0x00000000
kfdhdb.ub4spare[40]:                  0 ; 0x19c: 0x00000000
kfdhdb.ub4spare[41]:                  0 ; 0x1a0: 0x00000000
kfdhdb.ub4spare[42]:                  0 ; 0x1a4: 0x00000000
kfdhdb.ub4spare[43]:                  0 ; 0x1a8: 0x00000000
kfdhdb.ub4spare[44]:                  0 ; 0x1ac: 0x00000000
kfdhdb.ub4spare[45]:                  0 ; 0x1b0: 0x00000000
kfdhdb.ub4spare[46]:                  0 ; 0x1b4: 0x00000000
kfdhdb.ub4spare[47]:                  0 ; 0x1b8: 0x00000000
kfdhdb.ub4spare[48]:                  0 ; 0x1bc: 0x00000000
kfdhdb.ub4spare[49]:                  0 ; 0x1c0: 0x00000000
kfdhdb.ub4spare[50]:                  0 ; 0x1c4: 0x00000000
kfdhdb.ub4spare[51]:                  0 ; 0x1c8: 0x00000000
kfdhdb.ub4spare[52]:                  0 ; 0x1cc: 0x00000000
kfdhdb.ub4spare[53]:                  0 ; 0x1d0: 0x00000000
kfdhdb.acdb.aba.seq:                  0 ; 0x1d4: 0x00000000
kfdhdb.acdb.aba.blk:                  0 ; 0x1d8: 0x00000000
kfdhdb.acdb.ents:                     0 ; 0x1dc: 0x0000
kfdhdb.acdb.ub2spare:                 0 ; 0x1de: 0x0000
```

以下命令用于读取文件目录元数据：aunum 取自磁盘头的 kfdhdb.f1b1locn（见上面的输出）。

```
[grid@localhost ~]$ kfed read /dev/oracleasm/disks/ASMDISK1 aunum=2 blknum=1 | more
kfbh.endian:                          1 ; 0x000: 0x01
kfbh.hard:                          130 ; 0x001: 0x82
kfbh.type:                            4 ; 0x002: KFBTYP_FILEDIR
kfbh.datfmt:                          1 ; 0x003: 0x01
kfbh.block.blk:                       1 ; 0x004: T=0 NUMB=0x1
kfbh.block.obj:                       1 ; 0x008: TYPE=0x0 NUMB=0x1
kfbh.check:                  3975443128 ; 0x00c: 0xecf472b8
kfbh.fcn.base:                     4161 ; 0x010: 0x00001041
kfbh.fcn.wrap:                        0 ; 0x014: 0x00000000
kfbh.spare1:                          0 ; 0x018: 0x00000000
kfbh.spare2:                          0 ; 0x01c: 0x00000000
kfffdb.node.incarn:                   1 ; 0x000: A=1 NUMM=0x0
kfffdb.node.frlist.number:   4294967295 ; 0x004: 0xffffffff
kfffdb.node.frlist.incarn:            0 ; 0x008: A=0 NUMM=0x0
kfffdb.hibytes:                       0 ; 0x00c: 0x00000000
kfffdb.lobytes:                 2097152 ; 0x010: 0x00200000
kfffdb.xtntcnt:                       6 ; 0x014: 0x00000006
kfffdb.xtnteof:                       6 ; 0x018: 0x00000006
kfffdb.blkSize:                    4096 ; 0x01c: 0x00001000
kfffdb.flags:                         1 ; 0x020: O=1 S=0 S=0 D=0 C=0 I=0 R=0 A=0
kfffdb.fileType:                     15 ; 0x021: 0x0f
kfffdb.dXrs:                         19 ; 0x022: SCHE=0x1 NUMB=0x3
kfffdb.iXrs:                         19 ; 0x023: SCHE=0x1 NUMB=0x3
kfffdb.dXsiz[0]:             4294967295 ; 0x024: 0xffffffff
kfffdb.dXsiz[1]:                      0 ; 0x028: 0x00000000
kfffdb.dXsiz[2]:                      0 ; 0x02c: 0x00000000
kfffdb.iXsiz[0]:             4294967295 ; 0x030: 0xffffffff
kfffdb.iXsiz[1]:                      0 ; 0x034: 0x00000000
kfffdb.iXsiz[2]:                      0 ; 0x038: 0x00000000
kfffdb.xtntblk:                       6 ; 0x03c: 0x0006
```

```
kfffdb.break:                        60 ; 0x03e: 0x003c
kfffdb.priZn:                         0 ; 0x040: KFDZN_COLD
kfffdb.secZn:                         0 ; 0x041: KFDZN_COLD
kfffdb.ub2spare:                      0 ; 0x042: 0x0000
kfffdb.alias[0]:             4294967295 ; 0x044: 0xffffffff
kfffdb.alias[1]:             4294967295 ; 0x048: 0xffffffff
kfffdb.strpwdth:                      0 ; 0x04c: 0x00
kfffdb.strpsz:                        0 ; 0x04d: 0x00
kfffdb.usmsz:                         0 ; 0x04e: 0x0000
kfffdb.crets.hi:           32958036 ; 0x050: HOUR=0x14 DAYS=0x12 MNTH=0x9 YEAR=0x7db
kfffdb.crets.lo:         1372204032 ; 0x054: USEC=0x0 MSEC=0x28b SECS=0x1c MINS=0x14
kfffdb.modts.hi:           32958036 ; 0x058: HOUR=0x14 DAYS=0x12 MNTH=0x9 YEAR=0x7db
kfffdb.modts.lo:         1372204032 ; 0x05c: USEC=0x0 MSEC=0x28b SECS=0x1c MINS=0x14
kfffdb.dasz[0]:                       0 ; 0x060: 0x00
kfffdb.dasz[1]:                       0 ; 0x061: 0x00
kfffdb.dasz[2]:                       0 ; 0x062: 0x00
kfffdb.dasz[3]:                       0 ; 0x063: 0x00
kfffdb.permissn:                      0 ; 0x064: 0x00
kfffdb.ub1spar1:                      0 ; 0x065: 0x00
kfffdb.ub2spar2:                      0 ; 0x066: 0x0000
kfffdb.user.entnum:                   0 ; 0x068: 0x0000
kfffdb.user.entinc:                   0 ; 0x06a: 0x0000
kfffdb.group.entnum:                  0 ; 0x06c: 0x0000
kfffdb.group.entinc:                  0 ; 0x06e: 0x0000
kfffdb.spare[0]:                      0 ; 0x070: 0x00000000
kfffdb.spare[1]:                      0 ; 0x074: 0x00000000
kfffdb.spare[2]:                      0 ; 0x078: 0x00000000
kfffdb.spare[3]:                      0 ; 0x07c: 0x00000000
kfffdb.spare[4]:                      0 ; 0x080: 0x00000000
kfffdb.spare[5]:                      0 ; 0x084: 0x00000000
kfffdb.spare[6]:                      0 ; 0x088: 0x00000000
kfffdb.spare[7]:                      0 ; 0x08c: 0x00000000
kfffdb.spare[8]:                      0 ; 0x090: 0x00000000
kfffdb.spare[9]:                      0 ; 0x094: 0x00000000
kfffdb.spare[10]:                     0 ; 0x098: 0x00000000
kfffdb.spare[11]:                     0 ; 0x09c: 0x00000000
kfffdb.usm:                             ; 0x0a0: length=0
kfffde[0].xptr.au:                    2 ; 0x4a0: 0x00000002
kfffde[0].xptr.disk:                  0 ; 0x4a4: 0x0000
kfffde[0].xptr.flags:                 0 ; 0x4a6: L=0 E=0 D=0 S=0
kfffde[0].xptr.chk:                  40 ; 0x4a7: 0x28
kfffde[1].xptr.au:                    2 ; 0x4a8: 0x00000002
kfffde[1].xptr.disk:                  1 ; 0x4ac: 0x0001
kfffde[1].xptr.flags:                 0 ; 0x4ae: L=0 E=0 D=0 S=0
kfffde[1].xptr.chk:                  41 ; 0x4af: 0x29
kfffde[2].xptr.au:                    2 ; 0x4b0: 0x00000002
kfffde[2].xptr.disk:                  2 ; 0x4b4: 0x0002
kfffde[2].xptr.flags:                 0 ; 0x4b6: L=0 E=0 D=0 S=0
kfffde[2].xptr.chk:                  42 ; 0x4b7: 0x2a
kfffde[3].xptr.au:                   60 ; 0x4b8: 0x0000003c
kfffde[3].xptr.disk:                  1 ; 0x4bc: 0x0001
kfffde[3].xptr.flags:                 0 ; 0x4be: L=0 E=0 D=0 S=0
kfffde[3].xptr.chk:                  23 ; 0x4bf: 0x17
kfffde[4].xptr.au:                   60 ; 0x4c0: 0x0000003c
```

```
kfffde[4].xptr.disk:                  0 ; 0x4c4: 0x0000
kfffde[4].xptr.flags:                 0 ; 0x4c6: L=0 E=0 D=0 S=0
kfffde[4].xptr.chk:                  22 ; 0x4c7: 0x16
kfffde[5].xptr.au:                    3 ; 0x4c8: 0x00000003
kfffde[5].xptr.disk:                  2 ; 0x4cc: 0x0002
kfffde[5].xptr.flags:                 0 ; 0x4ce: L=0 E=0 D=0 S=0
kfffde[5].xptr.chk:                  43 ; 0x4cf: 0x2b
kfffde[6].xptr.au:           4294967295 ; 0x4d0: 0xffffffff
kfffde[6].xptr.disk:              65535 ; 0x4d4: 0xffff
kfffde[6].xptr.flags:                 0 ; 0x4d6: L=0 E=0 D=0 S=0
kfffde[6].xptr.chk:                  42 ; 0x4d7: 0x2a
…
kfffde[359].xptr.au:         4294967295 ; 0xfd8: 0xffffffff
kfffde[359].xptr.disk:            65535 ; 0xfdc: 0xffff
kfffde[359].xptr.flags:               0 ; 0xfde: L=0 E=0 D=0 S=0
kfffde[359].xptr.chk:                42 ; 0xfdf: 0x2a
```

2. 修复被覆盖的 ASM 元数据信息

以下命令用于保存磁盘头元数据：

```
[grid@localhost ~]$ kfed read /dev/oracleasm/disks/ASMDISK3 aunum=0 blknum=0
text=asmdisk3.txt
```

以下命令用于模拟磁盘头元数据被覆盖：

```
[grid@localhost ~]$ dd of=/dev/oracleasm/disks/ASMDISK3 if=/dev/zero bs=4096 count=1
```

以下命令用于验证磁盘头元数据是否被损坏：

```
[grid@localhost ~]$ kfed read /dev/oracleasm/disks/ASMDISK3 aunum=0 blknum=0
kfbh.endian:                          0 ; 0x000: 0x00
kfbh.hard:                            0 ; 0x001: 0x00
kfbh.type:                            0 ; 0x002: KFBTYP_INVALID
kfbh.datfmt:                          0 ; 0x003: 0x00
kfbh.block.blk:                       0 ; 0x004: T=0 NUMB=0x0
kfbh.block.obj:                       0 ; 0x008: TYPE=0x0 NUMB=0x0
kfbh.check:                           0 ; 0x00c: 0x00000000
kfbh.fcn.base:                        0 ; 0x010: 0x00000000
kfbh.fcn.wrap:                        0 ; 0x014: 0x00000000
kfbh.spare1:                          0 ; 0x018: 0x00000000
kfbh.spare2:                          0 ; 0x01c: 0x00000000
B7EEC200 00000000 00000000 00000000 00000000  [................]
  Repeat 255 times
KFED-00322: Invalid content encountered during block traversal:
[kfbtTraverseBlock][Invalid OSM block type][][0]
```

以下命令用于备份的文件恢复磁盘头元数据：

```
[grid@localhost ~]$ kfed write /dev/oracleasm/disks/ASMDISK3 aunum=0 blknum=0
text=asmdisk3.txt
```

以下命令用于读取恢复后的磁盘头元数据：

```
[grid@localhost ~]$ kfed read /dev/oracleasm/disks/ASMDISK3 aunum=0 blknum=0
text=new.txt
```

以下命令将验证磁盘头元数据和用于恢复的备份文件是否一致：

```
[grid@localhost ~]$ diff asmdisk3.txt new.txt
```

以下命令用于重新加载修复后的磁盘：

```
[grid@localhost ~]$ sqlplus "/as sysasm"

SQL*Plus: Release 11.2.0.2.0 Production on Sat Apr 7 22:55:02 2012

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.2.0 - Production
With the Automatic Storage Management option

SQL> alter diskgroup AVM_DG mount;

Diskgroup altered.
```

以下命令用于重新验证磁盘是否被成功加载：

```
[grid@localhost ~]$ asmcmd
ASMCMD> lsdg
State     Type   Rebal Sector Block  AU     Total   Free  Req_mir    Usable  Offline  Voting  Name
                                             _MB     _MB  _free_MB  _file_MB  _disks  _files
MOUNTED   EXTERN  N     512    4096 1048576  5192   4882       0      4882      0       N    AVM_DG/
MOUNTED   NORMAL  N     512    4096 1048576 28646  23277    1414     10931      0       N      DATA/
MOUNTED   EXTERN  N     512    4096 1048576  7106   7054       0      7054      0       N    FRA_DG/
```

3. 修复部分字段被损坏的 ASM 元数据信息

首先创建文件 asmdisk3.txt，插入需要修复的磁盘头元数据的部分字段，比如：

```
kfbh.hard:                          130 ; 0x001: 0x82
```

以下命令将使用编辑的文件修复磁盘头元数据的部分字段：

```
[grid@localhost ~]$ kfed merge /dev/oracleasm/disks/ASMDISK3 aunum=0 blknum=0
text=asmdisk3.txt
```

8.4　如何使用 AMDU 导出 ASM 文件

AMDU 是 Oracle 11*g* 引入的一个新工具，主要用途是从 ASM 磁盘中抽取元数据，生成格式化的元数据块布局报告，并且可以从磁盘组中抽取 ASM 文件，将其写到操作系统文件中。尽管这个工具是随 11*g* 版本发布的，但它也支持 ASM 10*g*，我们可以 Metalink 下载对应平台的 ADMU 工具。

下面简单介绍 AMDU 工具的一些常见使用场景。

1. 生成元数据块布局报告

以下命令用于生成磁盘组 DATA 的元数据块布局报告，这些报告存放在 AMDU 自动产生的目录 amdu_2012_04_07_21_54_22 下：

```
[grid@localhost ~]$ amdu -diskstring '/dev/oracleasm/disks/*' -dump 'DATA'
amdu_2012_04_07_21_54_22/
AMDU-00204: Disk N0001 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0001: '/dev/oracleasm/disks/ASMDISK1'
AMDU-00204: Disk N0002 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0002: '/dev/oracleasm/disks/ASMDISK2'
AMDU-00204: Disk N0004 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0004: '/dev/oracleasm/disks/ASMDISK4'
```

该目录会默认生成下面这组文件：

```
[grid@localhost ~]$ cd *_22
[grid@localhost amdu_2012_04_07_21_54_22]$ ls -l
total 156160
-rw-r--r-- 1 grid oinstall     17200 Apr  7 21:54 DATA.map
-rw-r--r-- 1 grid oinstall 159698944 Apr  7 21:54 DATA_0001.img
-rw-r--r-- 1 grid oinstall      9113 Apr  7 21:54 report.txt
```

其中，report.txt 是磁盘组所包含的每个磁盘的总体说明，其内容如下：

```
[grid@localhost amdu_2012_04_07_21_54_22]$ cat report.txt
--*-amdu-*-

******************************* AMDU Settings ********************************
ORACLE_HOME = /u01/app/grid/product/11.2.0/grid
System name:    Linux
Node name:      localhost.localdomain
Release:        2.6.18-194.el5
Version:        #1 SMP Mon Mar 29 20:06:41 EDT 2010
Machine:        i686
amdu run:       07-APR-12 21:54:22
Endianess:      1

--------------------------------- Operations ---------------------------------
       -dump DATA

------------------------------- Disk Selection -------------------------------
 -diskstring '/dev/oracleasm/disks/*'

------------------------------ Reading Control -------------------------------

------------------------------- Output Control -------------------------------

********************************* DISCOVERY **********************************

----------------------------- DISK REPORT N0001 ------------------------------
                Disk Path: /dev/oracleasm/disks/ASMDISK1
           Unique Disk ID:
               Disk Label:
     Physical Sector Size: 512 bytes
                Disk Size: 7640 megabytes
               Group Name: DATA
                Disk Name: DATA_0000
       Failure Group Name: DATA_0000
              Disk Number: 0
```

```
            Header Status: 3
       Disk Creation Time: 2011/09/18 20:20:28.533000
          Last Mount Time: 2012/03/16 13:06:34.527000
    Compatibility Version: 0x0b200000(11020000)
         Disk Sector Size: 512 bytes
         Disk size in AUs: 7640 AUs
         Group Redundancy: 2
      Metadata Block Size: 4096 bytes
                  AU Size: 1048576 bytes
                   Stride: 113792 AUs
      Group Creation Time: 2011/09/18 20:20:28.379000
  File 1 Block 1 location: AU 2
              OCR Present: NO

----------------------------- DISK REPORT N0002 ------------------------------
                Disk Path: /dev/oracleasm/disks/ASMDISK2
           Unique Disk ID:
               Disk Label:
     Physical Sector Size: 512 bytes
                Disk Size: 7640 megabytes
               Group Name: DATA
                Disk Name: DATA_0001
       Failure Group Name: DATA_0001
              Disk Number: 1
            Header Status: 3
       Disk Creation Time: 2011/09/18 20:20:28.533000
          Last Mount Time: 2012/03/16 13:06:34.527000
    Compatibility Version: 0x0b200000(11020000)
         Disk Sector Size: 512 bytes
         Disk size in AUs: 7640 AUs
         Group Redundancy: 2
      Metadata Block Size: 4096 bytes
                  AU Size: 1048576 bytes
                   Stride: 113792 AUs
      Group Creation Time: 2011/09/18 20:20:28.379000
  File 1 Block 1 location: AU 2
              OCR Present: NO

----------------------------- DISK REPORT N0003 ------------------------------
                Disk Path: /dev/oracleasm/disks/ASMDISK3
           Unique Disk ID:
               Disk Label:
     Physical Sector Size: 512 bytes
                Disk Size: 5192 megabytes
               Group Name: AVM_DG
                Disk Name: AVM_DG_0000
       Failure Group Name: AVM_DG_0000
              Disk Number: 0
            Header Status: 3
       Disk Creation Time: 2011/09/18 20:30:22.034000
          Last Mount Time: 2012/03/16 13:06:34.413000
    Compatibility Version: 0x0b200000(11020000)
         Disk Sector Size: 512 bytes
         Disk size in AUs: 5192 AUs
```

```
        Group Redundancy: 1
     Metadata Block Size: 4096 bytes
                 AU Size: 1048576 bytes
                  Stride: 113792 AUs
     Group Creation Time: 2011/09/18 20:30:21.882000
 File 1 Block 1 location: AU 2
             OCR Present: NO

----------------------------- DISK REPORT N0004 ------------------------------
               Disk Path: /dev/oracleasm/disks/ASMDISK4
          Unique Disk ID:
              Disk Label:
    Physical Sector Size: 512 bytes
               Disk Size: 6683 megabytes
              Group Name: DATA
               Disk Name: DATA_0002
      Failure Group Name: DATA_0002
             Disk Number: 2
           Header Status: 3
      Disk Creation Time: 2011/10/28 15:17:35.238000
         Last Mount Time: 2012/03/16 13:06:34.527000
   Compatibility Version: 0x0b200000(11020000)
        Disk Sector Size: 512 bytes
        Disk size in AUs: 6683 AUs
        Group Redundancy: 2
     Metadata Block Size: 4096 bytes
                 AU Size: 1048576 bytes
                  Stride: 113792 AUs
     Group Creation Time: 2011/09/18 20:20:28.379000
 File 1 Block 1 location: AU 2
             OCR Present: NO

----------------------------- DISK REPORT N0005 ------------------------------
               Disk Path: /dev/oracleasm/disks/ASMDISK5
          Unique Disk ID:
              Disk Label:
    Physical Sector Size: 512 bytes
               Disk Size: 6683 megabytes
              Group Name: DATA
               Disk Name: DATA_0003
      Failure Group Name: DATA_0003
             Disk Number: 3
           Header Status: 3
      Disk Creation Time: 2011/10/28 17:06:28.417000
         Last Mount Time: 2012/03/16 13:06:34.527000
   Compatibility Version: 0x0b200000(11020000)
        Disk Sector Size: 512 bytes
        Disk size in AUs: 6683 AUs
        Group Redundancy: 2
     Metadata Block Size: 4096 bytes
                 AU Size: 1048576 bytes
                  Stride: 113792 AUs
     Group Creation Time: 2011/09/18 20:20:28.379000
 File 1 Block 1 location: AU 0
```

```
              OCR Present: NO

----------------------------- DISK REPORT N0006 -----------------------------
                Disk Path: /dev/oracleasm/disks/ASMDISK6
           Unique Disk ID:
               Disk Label:
     Physical Sector Size: 512 bytes
                Disk Size: 7106 megabytes
               Group Name: FRA_DG
                Disk Name: FRA_DG_0000
       Failure Group Name: FRA_DG_0000
              Disk Number: 0
            Header Status: 3
       Disk Creation Time: 2011/09/18 20:29:41.536000
          Last Mount Time: 2012/03/16 13:06:34.642000
    Compatibility Version: 0x0b200000(11020000)
         Disk Sector Size: 512 bytes
         Disk size in AUs: 7106 AUs
         Group Redundancy: 1
      Metadata Block Size: 4096 bytes
                  AU Size: 1048576 bytes
                   Stride: 113792 AUs
      Group Creation Time: 2011/09/18 20:29:41.440000
  File 1 Block 1 location: AU 2
              OCR Present: NO

***************** Slept for 6 seconds waiting for heartbeats *****************

************************** SCANNING DISKGROUP DATA ***************************
            Creation Time: 2011/09/18 20:20:28.379000
         Disks Discovered: 4
               Redundancy: 2
                  AU Size: 1048576 bytes
      Metadata Block Size: 4096 bytes
     Physical Sector Size: 512 bytes
          Metadata Stride: 113792 AU
   Duplicate Disk Numbers: 0

---------------------------- SCANNING DISK N0001 -----------------------------
Disk N0001: '/dev/oracleasm/disks/ASMDISK1'
AMDU-00204: Disk N0001 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0001: '/dev/oracleasm/disks/ASMDISK1'
** HEARTBEAT DETECTED **
           Allocated AU's: 1414
                Free AU's: 6226
       AU's read for dump: 56
       Block images saved: 10006
        Map lines written: 56
          Heartbeats seen: 1
  Corrupt metadata blocks: 0
        Corrupt AT blocks: 0

---------------------------- SCANNING DISK N0002 -----------------------------
Disk N0002: '/dev/oracleasm/disks/ASMDISK2'
```

```
AMDU-00204: Disk N0002 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0002: '/dev/oracleasm/disks/ASMDISK2'
** HEARTBEAT DETECTED **
           Allocated AU's: 1413
                Free AU's: 6227
       AU's read for dump: 56
       Block images saved: 10262
        Map lines written: 56
          Heartbeats seen: 1
  Corrupt metadata blocks: 0
        Corrupt AT blocks: 0

---------------------------- SCANNING DISK N0004 -----------------------------
Disk N0004: '/dev/oracleasm/disks/ASMDISK4'
AMDU-00204: Disk N0004 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0004: '/dev/oracleasm/disks/ASMDISK4'
** HEARTBEAT DETECTED **
           Allocated AU's: 1276
                Free AU's: 5407
       AU's read for dump: 55
       Block images saved: 10257
        Map lines written: 55
          Heartbeats seen: 1
  Corrupt metadata blocks: 0
        Corrupt AT blocks: 0

---------------------------- SCANNING DISK N0005 -----------------------------
Disk N0005: '/dev/oracleasm/disks/ASMDISK5'
           Allocated AU's: 1266
                Free AU's: 5417
       AU's read for dump: 48
       Block images saved: 8464
        Map lines written: 48
          Heartbeats seen: 0
  Corrupt metadata blocks: 0
        Corrupt AT blocks: 0

------------------------- SUMMARY FOR DISKGROUP DATA -------------------------
           Allocated AU's: 5369
                Free AU's: 23277
       AU's read for dump: 215
       Block images saved: 38989
        Map lines written: 215
          Heartbeats seen: 3
  Corrupt metadata blocks: 0
        Corrupt AT blocks: 0

******************************* END OF REPORT ********************************
```

DATA.map 记录了磁盘组中元数据文件的扩展分配表及其在 img 文件中的位置，其内容如下：

```
[grid@localhost amdu_2012_04_07_21_54_22]$ cat DATA.map
N0001 D0000 R00 A00000000 F00000000 I0 E00000000 U00 C00256 S0001 B0000000000
N0001 D0000 R00 A00000001 F00000000 I0 E00000000 U00 C00256 S0001 B0001048576
```

```
N0001 D0000 R00 A00000002 F00000001 I0 E00000000 U00 C00256 S0001 B0002097152
N0001 D0000 R00 A00000003 F00000002 I0 E00000001 U00 C00256 S0001 B0003145728
N0001 D0000 R00 A00000005 F00000003 I0 E00000003 U00 C00256 S0001 B0004194304
N0001 D0000 R00 A00000006 F00000003 I0 E00000007 U00 C00256 S0001 B0005242880
…
N0005 D0003 R00 A00000916 F00000266 I1 E00000002 U00 C00001 S0001 B0159678464
N0005 D0003 R00 A00000941 F00000267 I1 E00000002 U00 C00001 S0001 B0159682560
N0005 D0003 R00 A00001118 F00000269 I1 E00000000 U00 C00002 S0001 B0159686656
N0005 D0003 R00 A00001143 F00000271 I1 E00000002 U00 C00001 S0001 B0159694848
```

DATA_0001.img 是磁盘组中元数据的 DUMP 映像，每个文件最大为 1 GB。

2. 通过文件号导出 ASM 文件

以下命令将导出磁盘组 DATA 中文件号为 257 的文件：

```
amdu -diskstring '/dev/oracleasm/disks/*' -dump 'DATA' -norep -nodir -extr DATA.257
-output sysaux.dbf
```

以下命令用于确认 257 号文件是否被导出：

```
[grid@localhost ~]$ ls -l sysaux.dbf
-rw-r--r-- 1 grid oinstall 817897472 Apr  7 22:10 sysaux.dbf
```

以下命令用于验证导出的数据文件是否正确：

```
[grid@localhost ~]$ dbfsize sysaux.dbf
Database file: sysaux.dbf
Database file type: file system
Database file size: 99840 8192 byte blocks
```

3. 通过文件名导出 ASM 文件

以下命令将生成磁盘组 DATA 的所有别名信息，D1.A49 从 DATA.map 中获取，其对应的文件号为 F00000006：

```
[grid@localhost ~]$ amdu -diskstring '/dev/oracleasm/disks/*' -dump 'DATA' -norep
-nodir -print DATA.D1.A49.B0.C256 > dir.txt
AMDU-00204: Disk N0001 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0001: '/dev/oracleasm/disks/ASMDISK1'
AMDU-00204: Disk N0002 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0002: '/dev/oracleasm/disks/ASMDISK2'
AMDU-00204: Disk N0004 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0004: '/dev/oracleasm/disks/ASMDISK4'
```

从上面命令生成的 dir.txt 中搜寻想要导出的别名及文件号，比如 SYSTEM，其文件号为 256：

```
[grid@localhost ~]$ vi dir.txt
kfade[0].entry.incarn:                1 ; 0x024: A=1 NUMM=0x0
kfade[0].entry.hash:         2097622492 ; 0x028: 0x7d072ddc
kfade[0].entry.refer.number: 4294967295 ; 0x02c: 0xffffffff
kfade[0].entry.refer.incarn:          0 ; 0x030: A=0 NUMM=0x0
kfade[0].name:                   SYSTEM ; 0x034: length=6
kfade[0].fnum:                      256 ; 0x064: 0x00000100
kfade[0].finc:                762214281 ; 0x068: 0x2d6e7789
kfade[0].flags:                      18 ; 0x06c: U=0 S=1 S=0 U=0 F=1
kfade[0].ub1spare:                    0 ; 0x06d: 0x00
kfade[0].ub2spare:                    0 ; 0x06e: 0x0000
```

最后通过以下命令导出磁盘组 DATA 中的文件号为 256 的 SYSTEM 数据文件：

```
amdu -diskstring '/dev/oracleasm/disks/*' -dump 'DATA' -norep -nodir -extr DATA.256
-output system.dbf
```

4. 磁盘头损坏的导出

以下命令可以在磁盘头损坏的情况下生成磁盘组 DATA 的元数据块布局报告：

```
[grid@localhost ~]$ amdu -diskstring '/dev/oracleasm/disks/*' -ba 'DATA' -ausize
1048576 -blksize 4096
amdu_2012_04_07_22_16_33/
AMDU-00204: Disk N0001 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0001: '/dev/oracleasm/disks/ASMDISK1'
AMDU-00204: Disk N0002 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0002: '/dev/oracleasm/disks/ASMDISK2'
AMDU-00204: Disk N0004 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0004: '/dev/oracleasm/disks/ASMDISK4'
```

5. 分配表损坏的导出

以下命令可以在磁盘分配表损坏的情况下生成磁盘组 DATA 的元数据块布局报告：

```
[grid@localhost ~]$ amdu -diskstring '/dev/oracleasm/disks/*' -ba 'DATA' -ausize
1048576 -blksize 4096 -fullscan
amdu_2012_04_07_22_18_38/
AMDU-00204: Disk N0001 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0001: '/dev/oracleasm/disks/ASMDISK1'
AMDU-00204: Disk N0002 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0002: '/dev/oracleasm/disks/ASMDISK2'
AMDU-00204: Disk N0004 is in currently mounted diskgroup DATA
AMDU-00201: Disk N0004: '/dev/oracleasm/disks/ASMDISK4'
```

第 9 章

理解数据块结构



老白曾在《Oracle 优化日记》一书中，通过一个使用 bbed 修复系统的示例对数据块结构做过一些介绍。不过这些年 DBA 群体对于数据块结构的研究越来越多，这是因为从 Oracle 10*g* 版本开始，Oracle 对 dul 工具设置了使用时间的限制，除非购买 Oracle 原厂的现场服务，否则很难找到这个工具。另外，随着数据拯救的需求越来越多，dul 注重数据一致性的风格，使得它在处理某些极端需求时显得无能为力。

实际上，编写一些数据拯救工具，对于每个具有一定编程能力的 DBA 来说都不是很难，目前他们最为缺乏的只是对数据块结构的一些基本了解。本章同样是由老储编写的，希望下面的内容对于想编写类似 dul 这种数据拯救工具的朋友有所帮助。

9.1 理解数据块头结构

数据块是数据库进行磁盘 I/O 和信息存储的最小单位，可以有多种大小：2 KB、4 KB、8 KB、16 KB 或 32 KB。数据库创建时需要指定默认的块大小，主要用于系统、UNDO 和 TEMP 表空间，而每个用户创建的表空间可以使用不同的块大小。不管块大小取多少，每个数据块的块头结构都是固定的，并且其中的 WORD（16 位）和 INTEGER（32 位）的字节顺序（Endian）和所处平台的字节顺序一致，这样做的好处是可以获得最佳的性能，但是不能保证数据库的跨平台移植。当然，如果两个异种平台的字节顺序是一致的，使用 Oracle 10*g* 以上的版本也是可以移植的，通过查询 V$TRANSPORTABLE_PLATFORM 视图可以了解不同平台之间的移植性。数据块头结构随版本的不同可能会有差异，我们主要接触到的有 V7 和 V8，8.0 以后的版本基本上保持着兼容性。数据块有多种类型，每种类型的结构也各不相同，但是所有的数据块都包含相同的头结构——kcbh。以 V8 的数据块为例，块头由 20 B 组成，位于块的起始部分，主要结构如下：

```
+-----+-----+-----+-------+--------+--------+------+-----+-------+-------+
|  0  |  1  | 2, 3|  4-7  |  8-11  |  12-13 |  14  | 15  | 16-17 | 18-19 |
+-----+-----+-----+-------+--------+--------+------+-----+-------+-------+
| 类型 | 格式 | 填充 | RDBA  |SCN BASE|SCN WRAP| 序列号 | 标志 | 校验和 |  填充  |
+-----+-----+-----+-------+--------+--------+------+-----+-------+-------+
```

下面按照顺序介绍每个字段的含义。

❑ 类型（type_kcbh）定义了该块的用途，主要有以下几种：

```
0  = none.
1  = KTU UNDO HEADER
2  = KTU UNDO BLOCK
3  = SAVE UNDO HDR
4  = SAVE UNDO BLOCK
5  = DATA SEG HDR
6  = KTB MANAGED (with ITL)
7  = TEMP DATA (no itl)
8  = SORT KEY
9  = SORT RUN
10 = SEG FREE LIST
11 = DATA FILE HDR
12 = DATA SEG HDR with Free List Groups
13 = Compatibility SEG
14 = unlimited undo segment header
15 = unlimited save undo segment header
16 = unlimited data segment header
17 = unlimited data segment header with flg blks
18 = extent map block
19 = backup set piece header
20 = backup set directory block
21 = control file block
22 = segment free list block with #blks in freelists
23 = bitmapped segment header
24 = bitmapped freelist block
25 = bitmap index block
26 = bitmap block
27 = LOB block
28 = funny undo block (for future use)
29 = bitmapped file space header
30 = bitmapped file space bitmap block
31 = temporary seg based index
32 = First level bitmap block
33 = second level bitmap block
34 = third level bitmap block
35 = Pagetable segment header block
36 = Pagetable extent map block
37 = System Managed Undo Extent Map Block
38 = System Managed Segment Header Block
39 = SPFILE backup block
40 = pagetable managed LOB block
41 = max value for block header validation
```

- 格式（type_kcbh）主要包括：

```
1 = Oracle7
2 = Oracle8 以上
```

- RDBA（rdba_kcbh）：该块的相对数据库地址，共 32 位，高 10 位表示相对文件号，低 22 位表示块号。
- SCN BASE（bas_kcbh）、SCN WRAP（wrp_kcbh）：对应着当前 REDO 生成的 SCN，SCN 由 SCN BASE 和 SCN WRAP 两部分组成。

9

- 序列号（seq_kcbh）：在相同的 SCN 下，该块可能会产生多次变化，使用序列号进行区分，如果 SCN 增长，序列号复位为 1。
- 标志（flg_kcbh）由一些位值组合而成（OR 操作），这些位值主要包括：

```
KCBHFNEW 0x01   // 新建块
KCBHFDLC 0x02   // 数据块延迟清洗推进 SCN 和序列号
KCBHFCKV 0x04   // 设置校验和
KCBHFTMP 0x08   // 临时块
```

- 校验和（chkval_kcbh）：可选，如果设置了初始化参数 DB_BLOCK_CHECKSUM，将会启用校验和检查数据块的一致性。

另外，数据块的最后 4 字节为块尾（tail），可以配合数据块头来验证数据块的一致性。它的结构为：

SCN BASE 的 2 位低字节+块类型+序列号

如果一个块被标识成软损坏，那么其块头的序列号为 0xff，标志为 0x00。

下面分别介绍字节顺序不同的平台上的数据块头。

- Linux 平台：小端，低位字节在低地址。

```
Block: 242    Offsets:    0 to   19
-----------------------------------
06020000 f2004000 688c0200 00000106
46bf0000
struct kcbh, 20 bytes                   @0
   ub1 type_kcbh                        @0         0x06        --表/索引块
   ub1 frmt_kcbh                        @1         0x02        --V8
   ub1 spare1_kcbh                      @2         0x00
   ub1 spare2_kcbh                      @3         0x00
   ub4 rdba_kcbh                        @4         0x004000f2  --小端，字节顺序反转
   ub4 bas_kcbh                         @8         0x00028c68
   ub2 wrp_kcbh                         @12        0x0000
   ub1 seq_kcbh                         @14        0x01
   ub1 flg_kcbh                         @15        0x06 (KCBHFDLC, KCBHFCKV)
   ub2 chkval_kcbh                      @16        0xbf46     --KCBHFCKV 表示需要校验和
   ub2 spare3_kcbh                      @18        0x0000
```

- AIX 平台：大端，高位字节在低地址。

```
Block: 122    Offsets:    0 to   19
-----------------------------------
06020000 0040007a 934e5bc1 0b950106
bd5a0000
struct kcbh, 20 bytes                   @0
   ub1 type_kcbh                        @0         0x06
   ub1 frmt_kcbh                        @1         0x02
   ub1 spare1_kcbh                      @2         0x00
   ub1 spare2_kcbh                      @3         0x00
   ub4 rdba_kcbh                        @4         0x0040007a  --大端，字节顺序无需反转
   ub4 bas_kcbh                         @8         0x934e5bc1
   ub2 wrp_kcbh                         @12        0x0b95
```

```
ub1 seq_kcbh                        @14       0x01
ub1 flg_kcbh                        @15       0x06 (KCBHFDLC, KCBHFCKV)
ub2 chkval_kcbh                     @16       0xbd5a
ub2 spare3_kcbh                     @18       0x0000
```

9.2　理解 ITL

事务型的数据块（类型为 6）存放表和索引数据，其结构主要由块头、事务层、数据层和块尾 4 部分组成。

其中，事务层由头结构和 ITL（Interested Transaction Slots）构成。ITL 是一个可变长的数组，当某个事务修改或准备修改（for update）该块中的行数据时，首先会在 ITL 中寻找一个空闲的插槽，如果没找到，就动态扩展 ITL，然后在空闲插槽中填写相关的事务信息。事务层头结构从块偏移 20 开始，其结构 ktbbh 为：

```
+-------+------+--------+------------+-------+------+----+-------+
|   0   | 1-3  |  4-7   |    8-15    | 16-17 |  18  | 19 | 20-23 |
+-------+------+--------+------------+-------+------+----+-------+
| 块类型 | 填充  | 对象段号 | 最后清洗 SCN | ITL 数 | 标志  | 锁  | 块指针 |
+-------+------+--------+------------+-------+------+----+-------+
```

下面分别介绍每个字段的含义。

- 块类型（ktbbhtyp）可取以下值：

```
1 = DATA
2 = INDEX
```

- 对象段号（ktbbhsid）是数据库对象所使用的段的编号，表示该块中的所有行数据属于该段。
- 最后清洗 SCN（ktbbhcsc）是该块最后一次执行清洗时的 SCN。
- ITL 数（ktbbhict）记录了 ITL 数组的长度。
- 标志（ktbbhflg）由一些位值组合而成（OR 操作），这些位值主要包括：

```
KTBFONFL       0x01  // 块在 Free List 中
KTBFBSEG       0x10  // 块使用位图管理空闲空间
KTBFEXTHD      0x20  // 支持扩展的事务头
```

- 锁（ktbbhfsl）是一个复用的字段，如果数据块采用位图管理空闲空间（ASSM 段），那么该字段表示该块所处位图数组的插槽号，如果数据块使用 Free List 管理空闲空间（FLM 段），那么该字段用于事务型 Free List 的操作。
- 块指针（ktbbhfnx）是另一个复用的字段，如果数据块采用位图管理空闲空间（ASSM 段），那么该字段表示 L1 块的块地址；如果数据块使用 Free List 管理空闲空间（FLM 段），那么该字段指向 Free List 中的下一个块。

ITL 数组的每个插槽由 24 个字节组成，从块偏移 44 或 52（如果使用位图管理）开始，其结构 ktbit 为：

```
+------+------+---------+--------+----+-----+--------+---------+
| 0-1  | 2-3  |   4-7   |  8-14  | 15 |16-17|  18-19 |  20-23  |
+------+------+---------+--------+----+-----+--------+---------+
|回滚段 |插槽号 |UNDO WRAP|UNDO 地址|保留 |标志  |SCN WRAP|SCN BASE |
+------+------+---------+--------+----+-----+--------+---------+
```

每个字段的含义如下所示。

- 回滚段（kxidusn）是对应事务所在的回滚段编号。
- 插槽号（kxidslt）是对应事务在回滚段事务表中的插槽号。
- UNDO WRAP（kxidsqn）是对应事务在回滚段事务表中的 WRAP 值，它和前两个字段一起唯一标识了一个事务，相当于事务的主键。
- UNDO 地址（ktbituba）是最后生成的 UNDO 记录的地址，格式为：

  UNDO DBA（4 字节）+序列号（2 字节）+记录号（1 字节）
- 如果某个行在事务中被连续改变了多次，每次都会生成 UNDO 记录，这些记录就被反向链接在一起，后一个指向前一个，这样在执行回滚时，可以从后往前逐个实施 UNDO 记录，最终回滚到某个保存点或事务的起点。
- 标志（ktbitflg）由两部分组成，高 4 位表示事务的提交状态，低 12 位保存该事务修改的本块的行数，事务的提交状态由一些位值组合而成（OR 操作），这些位值主要包括：

```
KTBFCOM        0x8000  // 事务已提交
KTBFIBI        0x4000  // 使用 ktbituba 指向的 UNDO 记录回滚可以获得本 ITL 插槽的前映像
KTBFUPB        0x2000  // 事务提交 SCN 是个上边界，不一定精确
KTBFTAC        0x1000  // 事务在最后清洗时（ktbbhcsc）尚未提交
```

- SCN WRAP（_ktbitun）：该字段复用，如果事务已提交并完成清洗，该字段保存事务提交 SCN 的 SCN WRAP 部分，否则该字段保存空闲空间预支字节数（Free Space Credit）。比如，删除一行可以释放 80 B 字节，在事务提交前，这 80 B 就属于 fsc，只有事务提交后，才能正式返回到空闲空间。
- SCN BASE（ktbitbas）：事务提交 SCN 的 SCN BASE 部分。

我们看一个例子：初始插槽为 2。

```
Linux 平台：小端，字节顺序需要反转
Block: 83362  Offsets:   20 to   91
-----------------------------------
01000000 59760000 df5be300 00000000
02000300 00000000 02000800 a5980000
43008000 84092500 07200000 a85ce300
00000000 00000000 00000000 00000000
00000000 00000000
struct ktbbh, 72 bytes                     @20        事务层头结构
   ub1 ktbbhtyp                            @20        0x01 (KDDBTDATA)
   union ktbbhsid, 4 bytes                 @24
      ub4 ktbbhsg1                         @24        0x00007659
      ub4 ktbbhod1                         @24        0x00007659
   struct ktbbhcsc, 8 bytes                @28
      ub4 kscnbas                          @28        0x00e35bdf
      ub2 kscnwrp                          @32        0x0000
```

```
b2 ktbbhict                         @36      0x0002         --两个 ITL 插槽
ub1 ktbbhflg                        @38      0x03 (KTBFONFL) --Free List 管理
ub1 ktbbhfsl                        @39      0x00
ub4 ktbbhfnx                        @40      0x00000000
struct ktbbhitl[0], 24 bytes        @44      第一个 ITL 插槽
   struct ktbitxid, 8 bytes         @44                          --事务标识
      ub2 kxidusn                   @44      0x0002
      ub2 kxidslt                   @46      0x0008
      ub4 kxidsqn                   @48      0x000098a5
   struct ktbituba, 8 bytes         @52                      --最后 UNDO 记录地址
      ub4 kubadba                   @52      0x00800043
      ub2 kubaseq                   @56      0x0984
      ub1 kubarec                   @58      0x25
   ub2 ktbitflg                     @60      0x2007(KTBFUPB)--该事务修改了 7 行
   union _ktbitun, 2 bytes          @62
      b2 _ktbitfsc                  @62      0
      ub2 _ktbitwrp                 @62      0x0000
   ub4 ktbitbas                     @64      0x00e35ca8
struct ktbbhitl[1], 24 bytes        @68      第二个 ITL 插槽
   struct ktbitxid, 8 bytes         @68
      ub2 kxidusn                   @68      0x0000
      ub2 kxidslt                   @70      0x0000
      ub4 kxidsqn                   @72      0x00000000
   struct ktbituba, 8 bytes         @76
      ub4 kubadba                   @76      0x00000000
      ub2 kubaseq                   @80      0x0000
      ub1 kubarec                   @82      0x00
   ub2 ktbitflg                     @84      0x0000 (NONE)     --该插槽未使用
   union _ktbitun, 2 bytes          @86
      b2 _ktbitfsc                  @86      0
      ub2 _ktbitwrp                 @86      0x0000
   ub4 ktbitbas                     @88      0x00000000
```


为什么要设置 INI_TRANS 参数

在创建表或索引时，可以指定 INI_TRANS 参数，该参数定义了表或索引数据块中预留的 ITL 插槽数。如果某个事务在 ITL 中没有找到空闲的插槽，就会动态扩展 ITL，这种扩展是从数据块的低端往高端进行的，正好和行数据相反。行数据是从高端往低端分配的，每个 ITL 插槽需要 24 B，如果数据块无足够的空闲空间，就会产生等待事件 enq: TX - allocate ITL entry。

当占用 ITL 插槽的某个事务提交或回滚后，释放了 ITL 插槽，正在等待的事务才能继续进行下去。这种现象一般出现在表上有很多并发事务执行 UPDATE 操作的情况下，解决的办法是增加表及索引的 INI_TRANS 值。默认情况下，表的 INI_TRANS 值是 2，索引的 INI_TRANS 值也是 2，索引自身的维护（节点分裂）是一个内部事务，需要 ITL 插槽，插槽 0 被保留用于索引维护内部事务。增加 INI_TRANS 值有两种方法。

- 创建表或索引时，指定 INITRANS 值，将对所有的数据块起作用。
- 表或索引被创建后，修改 INITRANS 值，只对修改后新建的数据块起作用。

INITRANS 值被指定为 3 时，ITL 插槽的初始分配情况如代码清单 9-1 所示：

代码清单 9-1

```
Linux 平台：小端，字节顺序需要反转
Block: 83362  Offsets:   20 to  115
-----------------------------------
01000000 59760000 998de300 00000000
03000300 00000000 02000a00 c1980000
3f008000 85092a00 01000000 00000000
03000000 c9990000 6b008000 d2090900
01000000 00000000 09000500 c3990000
26018000 83091d00 01000000 00000000
struct ktbbh, 96 bytes                  @20
   ub1 ktbbhtyp                         @20       0x01 (KDDBTDATA)
   union ktbbhsid, 4 bytes              @24
      ub4 ktbbhsg1                      @24       0x00007659
      ub4 ktbbhod1                      @24       0x00007659
   struct ktbbhcsc, 8 bytes             @28
      ub4 kscnbas                       @28       0x00e38d99
      ub2 kscnwrp                       @32       0x0000
   b2 ktbbhict                          @36       3              --ITL 插槽号为 3
   ub1 ktbbhflg                         @38       0x03 (KTBFONFL)
   ub1 ktbbhfsl                         @39       0x00
   ub4 ktbbhfnx                         @40       0x00000000
   struct ktbbhitl[0], 24 bytes         @44       第一个 ITL 插槽
      struct ktbitxid, 8 bytes          @44
         ub2 kxidusn                    @44       0x0002
         ub2 kxidslt                    @46       0x000a
         ub4 kxidsqn                    @48       0x000098c1
      struct ktbituba, 8 bytes          @52
         ub4 kubadba                    @52       0x0080003f
         ub2 kubaseq                    @56       0x0985
         ub1 kubarec                    @58       0x2a
      ub2 ktbitflg                      @60       0x0001 (NONE)
      union _ktbitun, 2 bytes           @62
         b2 _ktbitfsc                   @62       0
         ub2 _ktbitwrp                  @62       0x0000
      ub4 ktbitbas                      @64       0x00000000
   struct ktbbhitl[1], 24 bytes         @68       第二个 ITL 插槽
      struct ktbitxid, 8 bytes          @68
         ub2 kxidusn                    @68       0x0003
         ub2 kxidslt                    @70       0x0000
         ub4 kxidsqn                    @72       0x000099c9
      struct ktbituba, 8 bytes          @76
         ub4 kubadba                    @76       0x0080006b
         ub2 kubaseq                    @80       0x09d2
         ub1 kubarec                    @82       0x09
      ub2 ktbitflg                      @84       0x0001 (NONE)
      union _ktbitun, 2 bytes           @86
         b2 _ktbitfsc                   @86       0
         ub2 _ktbitwrp                  @86       0x0000
      ub4 ktbitbas                      @88       0x00000000
   struct ktbbhitl[2], 24 bytes         @92       第三个 ITL 插槽
```

```
struct ktbitxid, 8 bytes            @92
   ub2 kxidusn                      @92       0x0009
   ub2 kxidslt                      @94       0x0005
   ub4 kxidsqn                      @96       0x000099c3
struct ktbituba, 8 bytes            @100
   ub4 kubadba                      @100      0x00800126
   ub2 kubaseq                      @104      0x0983
   ub1 kubarec                      @106      0x1d
ub2 ktbitflg                        @108      0x0001 (NONE)
union _ktbitun, 2 bytes             @110
   b2 _ktbitfsc                     @110      0
   ub2 _ktbitwrp                    @110      0x0000
ub4 ktbitbas                        @112      0x00000000
```

9.3 理解记录结构

表中的数据以记录或行的形式保存在事务型的数据块（kcbh.type_kcbh=6 并且 ktbbh.ktbbhtyp=1）中，它们位于数据块的数据层。表记录的数据层包括头结构、表索引、行索引和行数据。

头结构在块中的起始位置可以按以下公式计算。

- 对于 Free List 管理空闲空间（FLM）的数据块：44+ITLs*24。
- 对于位图管理空闲空间（ASSM）的数据块：52+ITLs*24。

头结构 kdbh 由以下字段组成：

```
+-----+-----+-----+---------+-------+---------+---------+-------------+
|  0  |  1  | 2-3 |   4-5   |  6-7  |   8-9   |  10-11  |    12-13    |
+-----+-----+-----+---------+-------+---------+---------+-------------+
| 标志 | 表数 | 行数 |空闲行插槽|起始空间|  结束空间 |  空闲空间 |  最终空闲空间 |
+-----+-----+-----+---------+-------+---------+---------+-------------+
```

每项的含义如下所示。

- 表数（kdbhntab）定义了表索引数组的长度。
- 行数（kdbhnrow）定义了行索引数组的长度。
- 空闲行插槽（kdbhfrre）定义了行索引数组中第一个空闲的插槽号，如果没有空闲的，则为 0xFFFF。
- 起始空间（kdbhfsbo）定义了数据层中空闲空间的起始偏移。
- 结束空间（kdbhfseo）定义了数据层中空闲空间的结束偏移。
- 空闲空间（kdbhavsp）定义了数据层中空闲空间的字节数。
- 最终空闲空间（kdbhtosp）定义了 ITL 中的事务提交后，数据层中空闲空间的字节数。

表索引是一个数组，定义了行数据可能包括的表，由于簇可能包含多个表，这些表在物理存储上使用同一个段，它们相关的行（拥有相同的簇值）被放在一起，Oracle 在设计数据层的结构时考虑到了这一点，通过表索引来区分相关的行属于哪一个表。表索引的插槽由 4 个字节组成，其结构 kdbt 如下：

```
+-----+-----+
| 0-1 | 2-3 |
+-----+-----+
| 偏移 | 行数 |
+-----+-----+
```

每项的含义如下所示。

- 偏移（kdbtoffs）定义了该表在行索引中的起始插槽号。
- 行数（kdbtnrow）定义了该表在行索引中使用的插槽数。

对于非簇表，表索引只有一个插槽，而对于簇表，表索引可能有多个插槽，每个插槽对应簇中的一个表，对应关系为：插槽号=obj$表中的 tab#列。正因为如此，即使簇中的某个表没有对应的记录，在表索引中，也需要为它留一个插槽，行数为 0。

行索引也是一个数组，定义了该块中包含的所有行数据的位置。行索引的插槽由 2 个字节组成，其结构 kdbr 如下：

```
+-----+
| 0-1 |
+-----+
| 偏移 |
+-----+
```

偏移定义了该行数据在数据层中的偏移字节。在计算该行数据在块中的偏移值时，还要加上数据层自身的偏移值。

行数据由行头和各列值组成，行头的基本结构 kdrh 为：

```
+-----+------+-------+
|  0  |  1   |   2   |
+-----+------+-------+
| 标志 |  行锁 |  列数  |
+-----+------+-------+
```

每项的含义如下所示。

- 标志（kdrhflag）由一些位值组合而成（OR 操作），这些位值主要包括：

```
KDRHFK 0x80        // 簇的键
KDRHFC 0x40        // 簇表成员
KDRHFH 0x20        // 行头
KDRHFD 0x10        // 已删除
KDRHFF 0x08        // 第一个分片
KDRHFL 0x04        // 最后一个分片
KDRHFP 0x02        // 分片中的第一列延续自上一个分片
KDRHFN 0x01        // 分片中的最后一列将在下一分片继续
```

- 行锁（kdrhlock）是用修改该行的事务在 ITL 中的插槽号来表示的。
- 列数（kdrhccnt）定义了本行数据包含的列数。

在一般情况下，紧邻行头的就是各个列的数据，其结构为该列数据的长度和实际数据。列的长度为 1 ~ 3 B，如果第一个字节为 0xff，表示该列为空（NULL）；如果第一个字节为 0xfe，那么后两位表示字节的长度，否则第一个字节表示列数据的长度。列在数据行中的顺序一般和表定

义中的顺序一致，但也可能不完全相同，比如，表中的 LONG 字段在实际行数据中一般被放在最后，实际的列顺序在 col$表的 segcol#列中定义。

以下示例中的行数据来自表 TEST，结构为：

```
 Name                                      Null?    Type
 ----------------------------------------- -------- ---------------------------
 C1                                                 CHAR(500)
 C2                                                 CHAR(100)
 C3                                                 VARCHAR2(20)

Linux 平台：小端，字节顺序需要反转
Block: 83362  Offsets:  188 to  201
-----------------------------------
00010700 ffff2000 21006902 5a04
struct kdbh, 14 bytes                       @188      数据层头结构
   ub1 kdbhflag                             @188      0x00 (NONE)
   b1 kdbhntab                              @189      0x01       --包含 1 个表的数据
   b2 kdbhnrow                              @190      0x07       --包含 7 行数据
   sb2 kdbhfrre                             @192      0xffff     --行索引中无空闲插槽
   sb2 kdbhfsbo                             @194      0x0020     --空闲空间起始偏移
   sb2 kdbhfseo                             @196      0x0021     --空闲空间结束偏移
   b2 kdbhavsp                              @198      0x0269     --空闲空间长度
   b2 kdbhtosp                              @200      0x045a     --提交后空闲空间长度

Block: 83362  Offsets:  202 to  205
-----------------------------------
00000700
struct kdbt[0], 4 bytes                     @202     表索引
   b2 kdbtoffs                              @202      0x0000     --在行索引中起始偏移
   b2 kdbtnrow                              @204      0x0007     --在行索引中插槽数

Block: 83362  Offsets:  206 to  219
-----------------------------------
df0cd60c 7a04d90a dc08d308 2100
sb2 kdbr[0]                                 @206      0x0cdf             --第 0 行
sb2 kdbr[1]                                 @208      0x0cd6             --第 1 行
sb2 kdbr[2]                                 @210      0x047a             --第 2 行
sb2 kdbr[3]                                 @212      0x0ad9             --第 3 行
sb2 kdbr[4]                                 @214      0x08dc             --第 4 行
sb2 kdbr[5]                                 @216      0x08d3             --第 5 行
sb2 kdbr[6]                                 @218      0x0021             --第 6 行

Block: 83362  Offsets:  221 to  726
-----------------------------------
2c0001fe f4013720 20202020 20202020
20202020 20202020 20202020 20202020
…
20202020 20202020 20202020 20202020
20202020 20202020 2020
struct kdrh,  3 bytes                       @221      第 6 行数据，偏移：188+33 = 221
   ub1 kdrhflag                             @221      0x2c (KDRHFH,KDRHFF,KDRHFL)
```

```
   b1 kdrhlock                              @222      0x00
   b1 kdrhccnt                             @223      0x01              --包含 1 列数据
[COL 1]
   Length                                   @224      fe f4 01          --长度 500
   Data                                     @227      37 20 20  …       --值：'7'
```

从上面的 DUMP 信息可以看出，如果行数据的某一列（假定为第 *N* 列）后面的所有列取值都是 NULL，那么在行数据中只需要包含 *N* 列的数据，*N*+1 到最后一列可以忽略。在上例中，只有第一列 C1 有值，后面的 C2、C3 列都无值，所以 kdrhccnt=1，表示只包含一列的数据。

9.4　解析 Oracle 字段的内部数据存储格式

Oracle 表字段（列）支持的数据类型是很丰富的，既可以是简单类型，也可以是复杂类型，比如对象类型、内嵌表等。Oracle 支持的简单类型主要包括以下几种。

- 字符串类型：CHAR、NCHAR、VARCHAR2 和 NVARCHAR2。
- 数值类型：NUMBER。
- 二进制类型：RAW。
- 长字符串类型：LONG。
- 长二进制类型：LONG RAW。
- 日期类型：DATE。
- 时间戳类型：TIMESTAMP、TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE。
- 时间间隔类型：INTERVAL YEAR TO MONTH 和 INTERVAL DAY TO SECOND。
- 行地址类型：ROWID 和 UROWID。

下面逐一进行介绍。

1. CHAR

表示固定长度的字符串，长度以字节计数，最大为 2000，如果字符串的实际长度小于所定义的长度，则尾部用空格（0x20）填充，字符按创建数据库时指定的字符集解析。内部代码为 96。

```
select dump(cast('12345' as char(10))) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=96 Len=10: 49,50,51,52,53,32,32,32,32,32
```

2. VARCHAR2

表示可变长度的字符串，长度以字节计数，最大为 4000，由列长度字节指出字符串的实际长度，字符按创建数据库时指定的字符集解析。内部代码为 1。

```
select dump(cast('12345' as varchar2(10))) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=1 Len=5: 49,50,51,52,53
```

3. NCHAR

表示固定长度的字符串，长度以字符计数，最大为 2000，如果字符串的实际长度小于所定义的长度，则尾部用空格填充，字符按创建数据库时指定的国家字符集解析。内部代码为 96。

```
select dump(cast('12345' as nchar(10))) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=96 Len=20: 0,49,0,50,0,51,0,52,0,53,0,32,0,32,0,32,0,32,0,32
```

4. NVARCHAR2

表示可变长度的字符串，长度以字符计数，最大为 4000，由列长度字节指出字符串的实际长度，字符按创建数据库时指定的国家字符集解析。内部代码为 1。

```
select dump(cast('12345' as nvarchar2(10))) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=1 Len=10: 0,49,0,50,0,51,0,52,0,53
```

5. NUMBER

可以表示整数和浮点数，内部代码为 2。在原理上采用了一百进制的科学计数法表示数值类型的数据：

$$a\,\mathrm{E}\,n = a \times 100\text{^}n$$

其中，a 为有效数据，$1 \leqslant |a| < 100$；n 是幂，为整数。

比如，100.1 按这种方式可以表示为 1.0010 × 100^1。

具体来说，NUMBER 的内部表示由 3 部分组成：

(1) 符号位和幂字节，该字节包含了符号位和幂信息，可以分三种情况处理。

- 对于负数，该字节小于 128，并且幂=255 – 字节 – 128 – 65。
- 对于正数，该字节大于 128，并且幂=字节 – 128 – 65。
- 如果是零，那么该字节等于 128。

(2) 有效数字，每个字节表示 0 ~ 99 的两位数字，可以分两种情况处理。

- 对于负数，数字=101 – 字节。
- 对于正数，数字=字节 – 1。

(3) 冗余字节（102），只有负数才有，用于内部排序。

```
select dump(100.1) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=2 Len=4: 194,2,1,11

select dump(-100.1) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=2 Len=5: 61,100,101,91,102

select dump(0) dump from dual;
DUMP
```

```
--------------------------------------------------------------------------------
Typ=2 Len=1: 128

select dump(0.1001) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=2 Len=3: 192,11,2
```

6. DATE

由世纪、年、月、日、时、分和秒组成。前两个字节表示世纪和年，需要减去 100 获得实际的值，第 3、4 字节为月和日，最后 3 个字节表示时、分、秒，需要减去 1 获得实际的值。内部代码为 12。

```
select dump(cast(to_date('20120101','YYYYMMDD') as date)) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=12 Len=7: 120,112,1,1,1,1,1
```

7. TIMESTAMP

提供了比秒更精确的时间值，由世纪、年、月、日、时、分、秒和小数秒（1/1000000000）组成。前 7 个字节和 DATE 一致，后 4 个字节表示小数秒。内部代码为 180。

```
create table test(ts timestamp);
insert into test values(to_timestamp('20120101 000000.001','YYYYMMDD HH24MISS.ff'));
select dump(ts) dump from test;
DUMP
--------------------------------------------------------------------------------
Typ=180 Len=11: 120,112,1,1,1,1,1,0,15,66,64
```

8. TIMESTAMP WITH TIME ZONE

和 TIMESTAMP 类似，不过增加了时区信息。最后两位分别表示时区小时和时区分钟。时区小时需要减去 20 得到实际的值，而时区分钟需要减去 60 得到实际的值。内部代码为 181。

```
create table test(ts timestamp with time zone);
insert into test values(to_timestamp('20120101 000000.001','YYYYMMDD HH24MISS.ff'));
select dump(ts) dump from test;
DUMP
--------------------------------------------------------------------------------
Typ=181 Len=13: 120,111,12,31,17,1,1,0,15,66,64,28,60
```

9. TIMESTAMP WITH LOCAL TIME ZONE

和 TIMESTAMP 类似，但是隐含了时区信息，实际时区取当前会话的时区。内部代码为 231。

```
create table test(ts timestamp with local time zone);
insert into test values(to_timestamp('20120101 000000.001','YYYYMMDD HH24MISS.ff'));
select dump(ts) dump from test;
DUMP
--------------------------------------------------------------------------------
Typ=231 Len=11: 120,112,1,1,1,1,1,0,15,66,64
```

10. RAW

表示可变长度的二进制字节串，长度以字节计数，最大为 2000，由列长度字节指出字节串

的实际长度。内部代码为 23。

```
select dump(cast('3132333435' as raw(10))) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=23 Len=5: 49,50,51,52,53
```

11. LONG

VARCHAR2 的扩展，表示可变长度的字符串，长度以字节计数，最大可达 2 GB，由列长度字节指出字符串的实际长度，字符按创建数据库时指定的字符集解析。内部代码为 8。

12. LONG RAW

RAW 的扩展，表示可变长度的二进制字节串，长度以字节计数，最大可达 2 GB，由列长度字节指出字符串的实际长度。内部代码为 24。

13. ROWID

表示行数据的地址，主要有两种类型：

(1) 短型（受限型），由 6 个字节组成，前 4 个字节为数据块地址（DBA），后 2 个字节表示位于块中的第几行。内部代码为 69。

(2) 长型（扩展型），由 10 个字节组成，前 4 个字节为对象的段标识，紧邻的 4 个字节为数据块地址（DBA），后 2 个字节表示位于块中的第几行。内部代码为 69。


```
select dump(cast('AAAHZZAABAAAUWiAAA' as rowid)) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=69 Len=10: 0,0,118,89,0,65,69,162,0,0
```

14. UROWID

通用型 ROWID，是 ROWID 的一种特殊表示形式，一般用于索引组织表和外部表。UROWID 可以表示 3 种不同类型的 ROWID，由第一个字节指明。内部代码为 208。不同类型的 UROWID 格式不同，分别为：

(1) 物理 ROWID，由 4 位对象标识、2 位文件号、4 位块号、2 位行号构成。

(2) 逻辑 ROWID，由可选的 DBA、行号以及必选的主键构成。

(3) 远程 ROWID，由远程数据库的键值构成。

```
select dump(cast('AAAHZZAABAAAUWiAAA' as urowid)) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=208 Len=13: 1,0,0,118,89,0,1,0,1,69,162,0,0
```

15. INTERVAL YEAR TO MONTH

日期型的时间间隔，由两部分组成，前 4 个字节为年份的间隔，需要减去 0x80000000 获得实际的值，后 1 个字节为月份的间隔，需要减去 60 获得实际的值。内部代码为 182。

```
select dump(cast('01-01' as interval year to month)) dump from dual;
DUMP
--------------------------------------------------------------------------------
Typ=182 Len=5: 128,0,0,1,61
```

16. INTERVAL DAY TO SECOND

时间型的时间间隔，由 3 部分组成，前 4 个字节为天的间隔，需要减去 0x80000000 获得实际的值，紧临的 3 个字节分别为时、分、秒的间隔，需要减去 60 获得实际的值。最后 4 个字节为小数秒的间隔，需要减去 0x80000000 获得实际的值。内部代码为 183。

```
select dump(cast('01 01:01:01.01' as interval day to second)) dump from dual;
DUMP
----------------------------------------------------------------------------------
Typ=183 Len=11: 128,0,0,1,61,61,61,128,152,150,128
```

行链、行迁移为什么会影响访问性能

在执行插入或更改行的操作时，由于块的空闲空间所限，整行数据可能无法完全位于同一块中，这时就需要将该行数据切割成多个分片，每个分片占用一行，位于不同的数据块（特殊情况下也可能位于同一个块）。这些分片通过内部指针串在一起，组成一个完整的行，并且用第一个分片的地址（ROWID）来定位该行，这就是行链。如果第一个分片不包含任何列数据，只包含指向下一个分片的指针，这种情况就叫做行迁移。行链有两种分片方式：一种方式是列间分片，即每个分片都包含一个或多个完整的列数据；另一种方式是列内分片，即一个列的数据被分割成几个分片。列内分片最有可能发生在 LONG 字段上，因为 LONG 字段最大可以达到 2 GB，单个数据块肯定放不下，所以需要用到多个分片。Oracle 为了在数据结构上支持分片，在行头后面增加了一个可选的指针字段，指向下一个分片，配合标志位的设置，可以实现分片的串接。这个指针字段为：

```
+---+----+
|0-3| 4-5|
+---+----+
|DBA|行号|
+---+----+
```

其中：

- DBA 是下个分片所在的数据块地址，采用大端顺序表示。
- 行号表示下个分片在数据块中位于第几行，采用大端顺序表示。

下面介绍行迁移的实现过程。

- 第一个分片的行头标志字段包含 KDRHFH 和 KDRHFF 位值，行头的指针指向下个分片。
- 最后一个分片的行头标志字段只包含 KDRHFL 位值，无行头指针。
- 中间分片的行头标志字段无 KDRHFL 位值，行头的指针指向下个分片。
- 对于列内分片，前面分片的行头标志字段包含 KDRHFN 位值，行头的指针指向下个分片，后面分片的行头标志字段包含 KDRHFP 位值。

对于行链，因为第一个分片是空的，所以第一个分片的行头标志字段只包含 KDRHFH 位值，并且行头的指针指向下个分片，而第二个分片的行头标志字段包含了 KDRHFF 位值。

以下示例中的行数据来自表 TEST，结构为：

```
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
C1                                                 NUMBER
C2                                                 VARCHAR2(4000)
C3                                                 LONG
C4                                                 DATE

Linux 平台：小端，字节顺序需要反转
select dbms_rowid.rowid_block_number(rowid)
block#,dbms_rowid.rowid_row_number(rowid) row# from test;
--行地址位于 83363 块的第 0 行
    BLOCK#       ROW#
---------- ----------
     83363          0

Block: 83363  Offsets: 4080 to 4091
-----------------------------------
28010100 4145a200 0002c102
struct kdrh,  9 bytes                     @4080     第 1 个分片：83363 块的第 0 行
   ub1 kdrhflag                           @4080     0x28 (KDRHFH,KDRHFF)
   ub1 kdrhlock                           @4081     0x01
   ub1 kdrhccnt                           @4082     0x01            --包含 1 列数据
   struct kd4ssrid kdrhnrid               @4083     下个分片地址（标志无 KDRHFL 位值）
      krdba    dba_kd4ssrid               @4083     0xa004145a2     --块 83362
      kd_sno   sno_kd4ssrid               @4087     0x0000          --第 0 行
[COL 1]                                   表 TEST 第 1 列：C1
   Length                                 @4089     0x02            --长度 2
   Data                                   @4090     c1 02           --值：1

Block: 83362  Offsets:  193 to 216
-----------------------------------
01020100 4145a400 000e4331 31313131
31313131 31313131
struct kdrh,  9 bytes                     @193     第 2 个分片：83362 块的第 0 行
   ub1 kdrhflag                           @193     0x01 (KDRHFN)   --最后 1 列被分片
   ub1 kdrhlock                           @194     0x02
   ub1 kdrhccnt                           @195     0x01            --包含 1 列
   struct kd4ssrid kdrhnrid               @196     下个分片地址（标志无 KDRHFL 位值）
      krdba    dba_kd4ssrid               @196     0xa004145a4     --块 83364
      kd_sno   sno_kd4ssrid               @200     0x0000          --第 0 行
[COL 1]                                   表 TEST 第 2 列：C2 (未完)
   Length                                 @202     0x0e            --长度：14
   Data                                   @203     43 31 …        --值：'C1...'

Block: 83364  Offsets:  195 to  314
-----------------------------------
03020100 4145a600 006e3131 31313131
31313131 31313131 31313131 31313131
...
31313131 31313131
struct kdrh,  9 bytes                    @195   第 3 个分片：83364 块的第 0 行
   ub1 kdrhflag                          @195   0x03 (KDRHFP,KDRHFN) --最后 1 列被分片
   ub1 kdrhlock                          @196     0x02
   ub1 kdrhccnt                          @197      0x01                 --包含 1 列
```

```
   struct kd4ssrid kdrhnrid              @198     下个分片地址（标志无 KDRHFL 位值）
      krdba    dba_kd4ssrid              @198     0xa004145a6      --块 83366
      kd_sno   sno_kd4ssrid              @202     0x0000           --第 0 行
[COL 1]                                  表 TEST 第 2 列：C2 (紧接上一分片，未完)
   Length                                @204     0x6e             --长度：110
   Data                                  @205      31 31 ...       --值：'1...'

Block: 83366  Offsets:  195 to  4091
-----------------------------------
03010300 4145a500 00feb90e 31313131
31313131 31313131 31313131 31313131
...
31313131 31077870 01010101 016b4333
33333333 33333333 33333333 33333333
33333333 33333333 33
struct kdrh,  9 bytes                 @195      第 4 个分片：83366 块的第 0 行
   ub1 kdrhflag                       @195      0x03 (KDRHFP,KDRHFN) --最后 1 列被分片
   ub1 kdrhlock                       @196      0x01
   ub1 kdrhccnt                       @197      0x03                 --包含 3 列
   struct kd4ssrid kdrhnrid           @198      下个分片地址（标志无 KDRHFL 位值）
      krdba    dba_kd4ssrid           @198      0xa004145a5      --块 83365
      kd_sno   sno_kd4ssrid           @202      0x0000           --第 0 行
[COL 1]                               表 TEST 第 2 列：C2 (紧接上一分片，结束)
   Length                             @204      fe b9 0e         --长度：3769
   Data                               @207      31 31 ...         --值：'1...'
   列 C2 由 3 个分片组成，总长：14+110+3769=3893
[COL 2]                               表 TEST 第 4 列：C4（因为第 3 列 C3 为 LONG 字段，实际存
                                      储位置应该位于最后，所以原来的第 4 列 C4 被交换到前面）
   Length                             @3976     0x07             --长度：7
   Data                               @3977     78 70 01 01 01 01 01 --值：2012-01-01
[COL 3]                               表 TEST 第 3 列：C3 (未完)
   Length                             @3984     0x6b             --长度：107
   Data                               @3985     43 33 ...         --值：'C3...'

Block: 83365  Offsets:  193 to  4091
-----------------------------------
060101fe 350f3333 33333333 33333333
33333333 33333333 33333333 33333333
...
33333333 33333333 333333
struct kdrh,  3 bytes                 @193      第 5 个分片：83366 块的第 0 行
   ub1 kdrhflag                       @193      0x06 (KDRHFP,KDRHFL)  --最后 1 个分片
   ub1 kdrhlock                       @194      0x01
   ub1 kdrhccnt                       @195      0x01             --包含 1 列
[COL 1]                               表 TEST 第 3 列：C3 (紧接上一分片，结束)
   Length                             @196      fe 35 0f         --长度：3893
   Data                               @199      33 33 ...         --值：'33...'
   列 C3 由 2 个分片组成，总长：107+3893=4000
```

对于行链或行迁移，因为一个完整的行数据由多个分片构成，这些分片很可能位于不同的数据块，因此，在访问这些行数据时，需要执行多个逻辑读才能完成一行数据的访问。另外，对于 DML 操作，除了第一个分片外，涉及修改的其他分片也都需要加行锁，这些附加的 ITL 操作极大地影响了性能。

9.5　理解 LOB 的存储结构

大对象数据类型（LOB）是从 Oracle 8 开始出现的，分为内部存储 LOB 和外部存储 LOB，内部存储 LOB 可以保存二进制数据（BLOB）或字符数据（CLOB 和 NCLOB）。外部存储 LOB（BFILE）比较特殊，具体的数据保存在操作系统文件中，而文件路径被保存为 BFILE 列值。在 LOB 出现之前，大数据只能使用 LONG 或 LONG RAW 字段保存，但是 LONG 或 LONG RAW 类型存在以下缺陷。

- 需要和行数据一起保存，这样不仅增加了段的长度，并且容易形成行链接，加大了表的访问成本。
- 最大长度限制为 2 GB。
- 在一个表中只能有一个 LONG 字段。
- 字符集必须和数据库的本地字符集一致。
- 只能作为一个整体被读取到内存或转储到文件中才能对其内容进行操作，很不方便。

与 LONG 字段相比，LOB 有以下优势。

- 数据存放灵活，既可以保存在行中（不得超过 4000 B），也可以保存在单独的 LOB 段中。
- 从 Oracle 10*g* 版本开始，最大长度可以达到 2 ~ 128 TB（9*i* 版本限制为 4 GB）。
- 在一个表中可以有多个 LOB 字段。
- 字符集采用 UNICODE，可以支持多国字符。
- 支持随机访问，可以采用类似文件流的方式对其内容进行定位和操作，很方便。

LOB 数据有两种保存方式，如果数据量比较小（实际数据 + 额外开销≤4000 B），可以直接保存在行中；如果数据量比较大，或者在定义 LOB 列时，明确指定保存在行外，那么 LOB 将被保存在单独的 LOB 段中。不管是在行中还是在 LOB 段中，都要用到 LOB 定位器，这是 LOB 最重要的数据结构，保存了 LOB 的信息，其结构 kolbl 如下：

```
+-----+-----+-------+-----+-----+-----+---------+-------+-------+
| 0-1 | 2-3 |   4   |  5  |  6  |  7  |   8-9   | 10-19 |  20-  |
+-----+-----+-------+-----+-----+-----+---------+-------+-------+
| 长度 | 版本 | 标志 1 |标志 2| 保留 | 保留 |  字符宽度 |  标识  |I-NODE|
+-----+-----+-------+-----+-----+-----+---------+-------+-------+
```

下面分别介绍每项的含义。

- 长度（len）表示定位器的长度，一般固定为 0x54，不同于实际长度。
- 标志 1（lobflg1）主要包括以下几种取值：

```
KOLBLBLOB   0x00000001  // BLOB
KOLBLCLOB   0x00000002  // CLOB
```

- 标志 2（lobflg2）由一些位值组合而成（OR 操作），这些位值主要包括：

```
KOLBLIDX    0x00000004  // 包含 I-NODE
KOLBLINI    0x00000008  // 定位器已初始化
```

- 字符宽度（bytelen）表示 LOB 数据使用的字符集的字节长度，一般为 2（UNICODE 字符）。

9

- 标识（lobid）是该 LOB 的主键。
- I-NODE 包含了 LOB 数据的存储信息，它在定位器中是可选的字段，如果标志 2 设置了 KOLBLIDX，那么该字段就存在，否则就不存在。一般而言，如果 LOB 列在定义时包含了行内存储（enable strorage in row）选项，在定位器中就会包含 I-NODE；而如果包含了行外存储（disable strorage in row）选项，在定位器中就不会包含 I-NODE，这时 I-NODE 包含在第一个 LOB 索引项中。

接下来，我们来分析 I-NODE 的数据结构 kdlinode：

```
+-----+-----+-----+-----+-------+-------+-----+
| 0-1 |  2  |  3  | 4-7 |  8-9  | 10-15 | 16- |
+-----+-----+-----+-----+-------+-------+-----+
| 长度 | 标志 | 保留 | 块数 | 字节数 |  版本  | 数据 |
+-----+-----+-----+-----+-------+-------+-----+
```

每项的含义如下所示。

- 长度（size_kdlinode）表示整个 I-NODE 的长度（包括数据）。
- 标志（flag_kdlinode）由一些位值组合而成（OR 操作），这些位值主要包括：

```
KDLIDINI    0x00000001   // LOB 有效
KDLIDIDX    0x00000002   // I-NODE 位于 LOB 索引中
KDLIDDBA    0x00000004   // 包含数据是 CHUNK 的块地址
KDLIDDAT    0x00000008   // 包含数据是实际的列值
```

- 块数（blocks_kdlinode）是 LOB 数据所占用的完整 CHUNK 数。
- 字节数（bytes_kdlinode）是 LOB 数据在占用的最后一个不完整 CHUNK 中实际使用的字节数。
- 数据是可选字段，根据标志的取值解读其意义，它可能有以下几种情况。
  - 包含实际的数据，数据的长度为 I-NODE 长度 – 16。
  - 包含 CHUNK 的块地址（DBA），每个块地址需要 4 个字节，因此实际包含的 CHUNK 数为(I-NODE 长度 – 16) / 4，最多可以包含 12 个 CHUNK 的块地址。

对于包含大量 CHUNK 的 LOB 数据，不可能把它的所有 CHUNK 块地址都放在 I-NODE 中，因此 Oracle 又设计了 LOB 索引这样的结构，用来存放所有 CHUNK 的地址，LOB 索引的键为 LOB 标识 + CHUNK 号，一个索引项最多可以存放 8 个 CUHNK 的地址。

LOB 段是以 CHUNK 为单位组织数据的，每个 CHUNK 是一组连续的数据块，如果修改了 CHUNK 中的数据，即使是一个字节，也会生成一个新的 CHUNK，原来的 CHUNK 作为旧版本被保存，以支持一致性读操作。为了限制 CHUNK 所有旧版本占用的空间数量，在创建 LOB 列时，可以设置一个阈值，一旦超出该阈值，最早的一些旧版本会被删除。这样，在访问旧版本时，就可能导致 ORA-01555 错误。

LOB 数据结构（定位器和 I-NODE）的字节顺序和前面提到的一些数据结构不同，它与平台无关，统一采用大端字节顺序。另外，CLOB 列中的字符数据使用的字符集可以支持多国字符，在不同的版本中存在一些区别，见表 9-1。

表　9-1

<table>
<tr><th rowspan="2">DB 版本</th><th rowspan="2">平台字节顺序</th><th rowspan="2">DB兼容性模式</th><th colspan="3">CLOB/NCLOB内部存储</th></tr>
<tr><th colspan="2">LOB (永久性)</th><th>临时LOB</th></tr>
<tr><td rowspan="2">Oracle 8.1.x、9.0.x、9.2.x版本</td><td>大端或小端</td><td>＞8.0</td><td colspan="2">UCS2</td><td>UCS2</td></tr>
<tr><td>大端或小端</td><td>8.0</td><td colspan="2">不允许</td><td>UCS2</td></tr>
<tr><td rowspan="4">Oracle 10g及以上</td><td>大端（AIX、HP、Solaris……）</td><td>不限（10g中最低模式是9.2）</td><td colspan="2">AL16UTF16</td><td>AL16UTF16</td></tr>
<tr><td rowspan="3">小端（Windows、Linux）</td><td rowspan="2">10.0</td><td>从9.2升级的表</td><td>UCS2</td><td>AL16UTF16</td></tr>
<tr><td>新建的表</td><td>AL16UTF16</td><td>AL16UTF16</td></tr>
<tr><td>9.2</td><td colspan="2">UCS2</td><td>AL16UTF16</td></tr>
</table>

从 10*g* 版本开始，CLOB 列和 NCLOB 都使用 AL16UTF16 字符集。这个字符集和平台无关，每个字符是一个双字节（WORD），采用大端表示。而在 8*i* 或 9*i* 版本中，使用的则是 UCS2，这个字符集和平台有关，由平台决定双字节字符的大、小端。

```
Name                                       Null?    Type
------------------------------------------ -------- ----------------------------
C1                                                  NUMBER
C2                                                  CLOB

Linux 平台：小端，数据库版本为 9.2，因此 CLOB 字符集采用 UCS2，也是小端，需要反转
Block: 83362  Offsets: 4049 to 4091
-----------------------------------
2c010202 c1022400 54000102 0c800000
02000000 01000000 004fb100 10090000
00000000 00000000 000000
struct kdrh,  3 bytes                        @4049
   ub1 kdrhflag                              @4049     0x2c (KDRHFH,KDRHFF,KDRHFL)
   ub1 kdrhlock                              @4050     0x01
   ub1 kdrhccnt                              @4051     0x02       --包含 2 列
[COL 1]
   Length                                    @4052     0x02       --长度：2
   Date                                      @4053     c1 02      --值：1
[COL 2]
   Length                                    @4055     0x24       --长度：36
   struct kolbl                              @4056     值为定位器
      ub2 len                                @4056     0x0054
      ub2 version                            @4058     0x0001
      ub1 lobflg1                            @4060     0x02 (KOLBLCLOB)
      ub1 lobflg2                            @4061     0x0c (KOLBLIDX, KOLBLINI)
      ub1 lobflg3                            @4062     0x80
      ub1 lobflg4                            @4063     0x00
      ub2 bytelen                            @4064     0x0002
      ub1 lobid[10]                          @4066     00 00 00 01 00 00 00 00 4f b1
struct kdlinode
   ub2 size_kdlinode                         @4076     0x0010    --I-NODE 长度 16
   ub1 flag_kdlinode                         @4078     0x09 (KDLIDINI, KDLIDDAT)
   ub1 future_kdlinode                       @4079     0x00
```

```
   kdlpage block_kdlinode              @4080     0x00000000
   k_blk bytes_kdlinode                @4084     0x0000
   kdlvn version_kdlinode              @4086     0x0000.00.00.00.00
   该 LOB 无实际数据，是由 empty_clob()产生的

Block: 83362  Offsets: 3996 to 4048
-----------------------------------
2c020202 c1032e00 54000102 0c800000
02000000 01000000 004fb200 1a090000
00000000 0a000000 00000131 00320033
00340035 00
struct kdrh,  3 bytes                  @3996
   ub1 kdrhflag                        @3996     0x2c (KDRHFH,KDRHFF,KDRHFL)
   ub1 kdrhlock                        @3997     0x02
   ub1 kdrhccnt                        @3998     0x02      --包含 2 列
[COL 1]
   Length                              @3999     0x02      --长度 2
   Date                                @4000     c1 03     --值：2
[COL 2]
   Length                              @4002     0x2e      --长度：46
   struct kolbl                        @4003     值为定位器
      ub2 len                          @4003     0x0054
      ub2 version                      @4005     0x0001
      ub1 lobflg1                      @4007     0x02 (KOLBLCLOB)
      ub1 lobflg2                      @4008     0x0c (KOLBLIDX, KOLBLINI)
      ub1 lobflg3                      @4009     0x80
      ub1 lobflg4                      @4010     0x00
      ub2 bytelen                      @4011     0x0002
      ub1 lobid[10]                    @4013     00 00 00 01 00 00 00 00 4f b2
struct kdlinode
   ub2 size_kdlinode                   @4023     0x001a   --I-NODE 长度 26
   ub1 flag_kdlinode                   @4025     0x09 (KDLIDINI, KDLIDDAT)
   ub1 future_kdlinode                 @4026     0x00
   kdlpage block_kdlinode              @4027     0x00000000
   k_blk bytes_kdlinode                @4031     0x000a
   kdlvn version_kdlinode              @4033     0x0000.00.00.00.01
   Data                                @4039     31 00 32 00 33 00 34 00 35 00
   该 LOB 实际数据为：'12345'

Block: 83362  Offsets: 3949 to 3995
-----------------------------------
2c010202 c1042800 54000102 0c800000
02000000 01000000 004fe300 14050000
0000000f a0000000 00000200 4145b4
struct kdrh,  3 bytes                  @3949
   ub1 kdrhflag                        @3949     0x2c (KDRHFH,KDRHFF,KDRHFL)
   ub1 kdrhlock                        @3950     0x01
   ub1 kdrhccnt                        @3951     0x02      --包含 2 列
[COL 1]
   Length                              @3952     0x02      --长度：2
   Date                                @3953     c1 04     --值：3
[COL 2]
   Length                              @3955     0x28      --长度：40
   struct kolbl                        @3956     值为定位器
```

```
      ub2 len                          @3956     0x0054
      ub2 version                      @3958     0x0001
      ub1 lobflg1                      @3960     0x02 (KOLBLCLOB)
      ub1 lobflg2                      @3961     0x0c (KOLBLIDX, KOLBLINI)
      ub1 lobflg3                      @3962     0x80
      ub1 lobflg4                      @3963     0x00
      ub2 bytelen                      @3964     0x0002
      ub1 lobid[10]                    @3966     00 00 00 01 00 00 00 00 4f e3
struct kdlinode
   ub2 size_kdlinode                   @3976     0x0014   --I-NODE 长度 20
   ub1 flag_kdlinode                   @3978     0x05 (KDLIDINI, KDLIDDBA)
   ub1 future_kdlinode                 @3979     0x00
   kdlpage block_kdlinode              @3980     0x00000000
   k_blk bytes_kdlinode                @3984     0x0fa0  --最后一个 CHUNK 包含 4000 字节
   kdlvn version_kdlinode              @3986     0x0000.00.00.00.02
   Data                                @3992     00 41 45 b4 --CHUNK 的块地址
   该 LOB 实际数据位于块 83380（LOB 段），包含 2000 个字符（4000 字节），实际数据如下：
Block: 83380   Offsets:  56 to 4055
----------------------------------- ----------------
58005800 58005800 58005800 58005800  X.X.X.X.X.X.X.X.
...
58005800 58005800 58005800 58005800  X.X.X.X.X.X.X.X.
```

LOB 字段的优化

通过以上对 LOB 字段存储结构的介绍，我们可以总结出以下一些有用的经验。

- 如果应用在查询表时，LOB 列和其他列是分开访问的，可以考虑将所有的 LOB 列数据保存在 LOB 段，也就是设置行外存储（disable strorage in row）选项。
- 如果 LOB 列的绝大部分数据比较大，可以设置较大的 CHUNK。
- 如果 LOB 列被频繁访问，可以考虑把它缓存到 DB Cache 中。默认情况下，LOB 是不被缓存的，需要通过设置 CACHE 选项打开，CACHE 也可以细分为缓存读写和只缓存读。
- 在不影响业务的情况下尽可能设置较小的版本阈值和保留期限，以避免过多的空间浪费。
- 对于需要在 LOB 字段上频繁执行一系列小的读写操作的应用，可以使用 LOB Buffering Subsystem，它会在客户端缓存、批量修改 LOB 字段的内容。
- 尽可能采用随机访问特性访问特定位置的内容，而不是所有内容。
- 在查询语句中，慎用“*”通配符选择所有列，只访问应用需要用到的列。

第 10 章 理解表的结构

10

表是我们最常见的对象，也是数据库中最为重要的对象。不过很多工作多年的 DBA 仍然不太了解表到底是什么。在本章中，老白将为大家“扫盲”，通过介绍表的内部存储结构，帮助大家更加深入地了解表存储优化方面的一些技巧。

10.1 到底什么是“表”

在本节中，我们要学习的内容是“表”。Oracle 数据库是一种关系型数据库，在关系型数据库里，一组关系的实体化就可以组成一张表。表包含一个或者多个字段，这些字段按照某种规则（我们称为关系）组成一个集合，每个关系就称为一行数据，也就是我们常说的一条记录。实际上，只要有了关系型数据库的概念，那么就不难理解表、记录、关系之类的术语了。表是 Oracle 数据库中十分重要的对象，DBA 的主要工作就是和不同的表和数据打交道。

Oracle 中的表是怎么存储的呢？想要弄清这个问题，就需要学习一些 Oracle 数据存储的基本概念。首先要了解什么是块（block），块是 Oracle 访问的最小物理单位，Oracle 数据库中块的大小一般是 2～32 KB。Oracle 的块大小和操作系统的块大小是不同的，前者往往远大于后者，因此，Oracle 的块不是基本的 I/O 单位，而只是 Oracle 的基本访问单位。对于 Oracle 来说，每个表空间都有唯一的块大小，在 Oracle 8*i* 或者更早版本的数据库中，整个数据库的所有表空间都有唯一的块大小。从 Oracle 9*i* 开始，不同的表空间可以拥有不同的块大小，因此参数 DB_BLOCK_SIZE 定义的就不再是数据库块大小，而是默认为表空间块大小。

虽然块是 Oracle 的最小访问单位，但是由于块太小了，如果表的存储数据按照块来分配，那么效率就太低了。这里继续介绍一个更大的单位——EXTENT，中文可以翻译成扩展，一个扩展是由一组连续的块组成的。扩展是 Oracle 最小的分配单位，Oracle 在给对象分配空间时，最少分配一个扩展。每个扩展是由一个或者多个连续的块组成的。这种连续是指文件上的连续，在目前存储技术充分使用条带化技术的前提下，这是一种逻辑上的连续，但在物理磁盘上并不一定就是连续的。

比扩展更大的单位是段（segment），一组结构相同的扩展就组成了一个段。最常见的段包括表、索引、表分区、索引分区、回滚段等。段存储在某个表空间中，表空间是个逻辑的概念，一个表空间一般包含一个或者多个数据文件，当然也可以不包含任何文件，这样的表空间是不能使

用的。实际上，段是存储在数据文件里的，但是这种存储是按照表空间来组织的。一个段可能存储在一个数据文件中，也可能存储在多个数据文件中，不过这些数据文件必须属于同一个表空间。图 10-1 是一个很好的逻辑示意图。

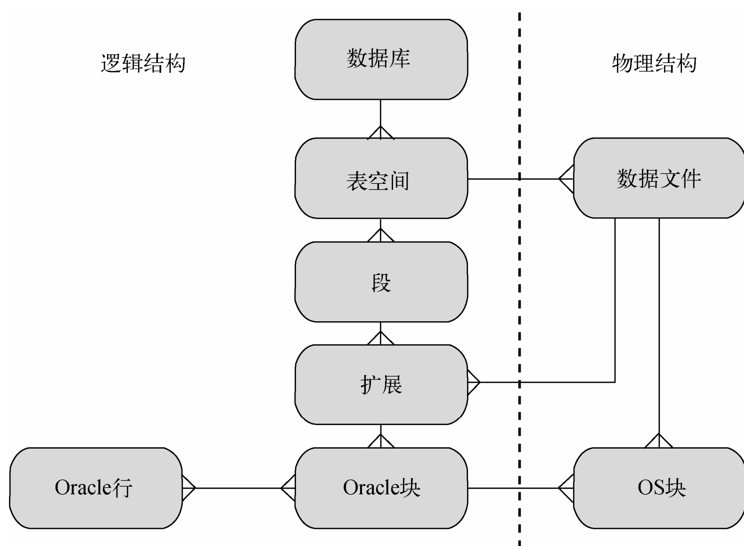


图 10-1

10

表在 Oracle 数据库中就是存储在段里的。一张普通的表就对应一个段，不过还有一些特例：

- 一张索引组织表可能包含了多个段，比如 OVERFLOW 段。
- 在一张表中，如果有 LOB 字段，LOB 字段可能存储在独立的段中，那么这张表就可能包含多个段。
- 一张分区表可能存储在多个独立的段中，这些段甚至可能存储在不同的表空间中。
- 在一个簇（cluster）中，可能存储多张表，而不是一张表。因此对于存储于簇中的表来说，表和段不是一一对应的。

通过上面的介绍，大家知道了表的数据实际上是以“段”的形式存储在表空间里的。当创建一个表的时候，Oracle 会自动为这张表创建段，并分配第一个扩展。我们已经知道了每个扩展是由多个连续的块组成的，数据就存储在这些块中。当这些块都使用完了，那么下一次需要插入数据的时候，Oracle 就会自动为这张表增加一个扩展。随着数据量的增长，这张表也在不断地增大。一张表中扩展的数量受到表空间容量和 MAXEXTENTS 参数的限制。对于早期版本的数据库，MAXEXTENTS 的默认值是 255，在这种情况下，某些较大的表很容易就会达到 MAXEXTENTS 的极限，这个时候再插入数据，可能就会出现 ORA-01631: Max # Extents (%s) Reached in TABLE %s.%s 错误信息。这时，就必须要加大 MAXEXTENTS 参数了。

在了解了表和段后，下面来分析表的内部。我们知道表中的数据是存储在块里的，那么块的结构是什么样的呢？

实际上，在一个块中，块头存储了一些数据块的信息、SCN、事务槽等信息，尾部的 4 个字节是块尾，块尾存在的目的是为了和块头的数据进行校验，确保这个数据块是一致的（具体的算法对于大多数人来说是不需要了解的，大家可以先放一放，在今后的进阶内容中再学习）。块头和块尾中间的部分就是存储的数据，如图 10-2 所示。表中的数据是从数据块的底部开始存储的，比如，某个块中存储了 3 条记录，那么第一条记录的尾部正好和块尾相接，存储在块的最后，第三条记录存储在靠近块头的地方，前面就是这个块的空闲空间。随着数据块中数据的增长，数据区域逐渐接近块头的位置，块中空闲的部分越来越少。如果这个块的空闲空间不足以插入一条新记录了，系统就会寻找别的可以插入数据的块；如果所有的数据块都已经满了，那么系统就会增加一个新的扩展。表中的空闲块管理，是个十分复杂的话题，我们将在后面专门讨论，对此还有疑问的朋友，就先收起这个疑问，继续学习其他的内容吧。Oracle 的知识点相当广，而且关联性十分强，在学习的时候，不一定非要一次性把所有的知识点都搞清楚，循序渐进才是最关键的，请大家不要心急。

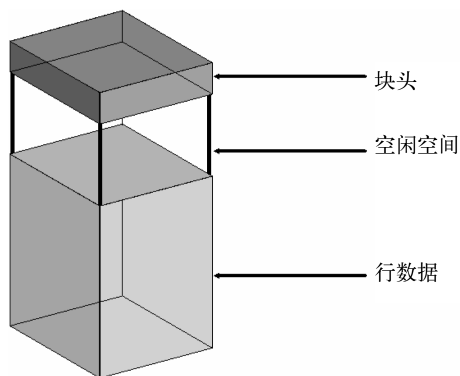


图　10-2

现在的表中都设计了大量的 VARCHAR 字段，VARCHAR 字段使用起来十分灵活，而且不会浪费空间。不过，VARCHAR 字段的使用也给块中的数据重组带来了麻烦。比如，在一张包含 VARCHAR 字段的表中插入一条数据的时候，如果首先插入了某个字段的 1 个字节，而后来通过 UPDATE 将这个字段的值修改为 200 个字节，那么这条记录就需要重组，从而导致整个块进行重组。而如果此时修改的这条记录所在的块已经没有 200 个空闲的字节了，那么这条记录修改后就无法存储在这个块中。Oracle 在处理这种情况的时候，会把这条记录整个迁移到另外一个存在足够空闲空间的数据块中，而在原来记录的地方登记一个指向新位置的指针，这种情况就是我们常说的行链。行链的产生，会加大系统的开销，影响数据访问的性能。

前面我们介绍了很多的技术细节，可能有些人已经昏昏欲睡了。实际上，下面的内容才是我们要学习的精华。对于一张表，我们在设计的时候要注意些什么呢？从上述关于段的知识中，我们可以得到以下的结论。

在创建表时，选择合适的扩展大小是十分重要的。过小的扩展会导致系统不断增加扩展的数

量，从而影响大数据量插入的性能，这对于存在大量并发插入的系统尤为重要。不过也不需要过于担心扩展大小所带来的性能问题，这里讨论的性能下降可能只是千分之一，甚至更小，实际上，一个段拥有小于 1024 个扩展对于段的访问性能影响很小。

下一个问题就是表空间碎片的问题。由于表空间中表的扩展大小不同，因此在使用一段时间后，表空间就会出现不连续的现象，在扩展之间会出现一些“洞”，这种情况我们一般叫做表空间碎片。一些经验丰富的 DBA 总是告诫别人，注意整理表空间碎片，否则会影响数据库的性能。实际上，从根本上来说，表空间碎片最大的危害是浪费了空间，而不是影响了性能。所有的数据访问都会在有效的扩展中进行，根本不会去扫描碎片所在的数据块，所以认定表空间碎片会影响性能是一种以讹传讹的观点。

事实上，对于表的参数定义来说，设置合理的 PCTFREE 值比减少表空间碎片重要得多。在老白碰到的客户中，能够根据表和业务的特点来设计 PCTFREE 的少之又少。不合理的 PCTFREE 可能导致大量行链的出现，从而影响访问性能。

在创建表的时候，还有一个十分重要的参数——INITRANS，这个参数指定了初始化事务槽的数量。事务槽是一个非常重要的对象，每个事务在修改某个数据块时，都需要在这个数据块中分配一个事务槽。如果当前没有空闲的事务槽，就需要动态扩展一个，每个事务槽需要 24 B。讲到事务槽，大家可能就明白 Oracle 为什么要从底部向顶部分配空间了，因为事务槽是从顶部向底部分配空间的，这样两种分配方式才不会产生冲突。一般情况下，默认的事务槽参数并不会带来明显的性能问题，不过对于一些并发修改较大的表，如果 PCTFREE 参数设置过低，就会导致事务槽扩展的时候无法分配空间，从而导致事务等待事务槽。我们可以通过下列脚本来检查系统中哪些表产生了较为严重的事务槽等待。

```
set line 132
col statistic_name format a30 trunc
SELECT t.OWNER, t.OBJECT_NAME, t.OBJECT_TYPE, STATISTIC_NAME, t.VALUE value
     FROM v$segment_statistics t
     WHERE t.STATISTIC_NAME = 'ITL waits'
     AND t.VALUE > 10 order by value;
```

如果我们通过上面的查询发现了事务槽等待较为严重的段，那么就需要考虑对这些段增加 INITRANS 参数值。不过由于这里查询到的值是数据库启动以来总的统计值，因此需要在一个时间段中多次执行这个 SQL，才能判断某个段是否真的经常出现 ITL 等待。在修改 INITRANS 参数的时候，我们也要注意，一旦修改了这个参数，对应段中新增的数据块会增加初始化事务槽的数量，而旧的数据块是不会改变的，如果要彻底解决问题，还需要对这个段进行一次重组。

10.1.1　PCTFREE 和行链

上一节介绍了表的基本结构，从表的存储结构上讨论了几个建表的参数。实际上，在许多项目中，开发团队并没有对表进行合理的设计，从而导致系统上线后出现大量的性能问题。

一个最常见的问题就是行链和行迁移。行链的出现在绝大多数系统中是由于设计不合理造成的，比如，某张表的行长度超过了一个数据块的大小，那么这个表的部分行就会出现行链。这种

行链实际上是由于设计者选择了不合理的数据块大小所导致的。从 Oracle 9*i* 开始，不同的表空间可以使用不同的块大小，因此我们完全可以设计块大小较大的表空间来存放这些表。

有些行链是不可避免的，而绝大多数的行迁移是可以避免的。行迁移是怎么产生的呢？Oracle 数据库支持 VARCHAR 字段，VARCHAR 字段的使用极为灵活，正是这种灵活性导致了行迁移的产生。在创建一张表的时候，我们一般不太会关注 PCTFREE 这个参数，该参数的默认值是 10%，其含义是，当某个数据块的使用率小于“100% – PCTFREE”时，这个数据块是可以插入数据的，一旦块使用率达到了这个指标，这个数据块就不能再插入数据了。Oracle 预留这部分空间的目的就是为 VARCAHR 字段的扩展提供空间。大家是不是还记得，Oracle 的数据块是从块的底部开始使用的，空闲空间在块头和数据之间。

下面通过一个示例来说明表数据的存储。首先创建一张测试表，并且插入三条记录：

```
create table test1 ( a integer,b varchar2(100),c varchar2(100));
insert into test1 values (1,null,'aaaa');
insert into test1 values (2,null,'bbbb');
insert into test1 values (1,'11111',null);
commit;
```

然后查找这个扩展所在的位置：

```
SQL> select extent_id,file_id,block_id from dba_extents where segment_name='TEST1'
AND OWNER='SCOTT';

 EXTENT_ID    FILE_ID   BLOCK_ID
---------- ---------- ----------
         0         10        497
```

这里，先查找 10 号文件是什么：

```
SQL>select file_name from dba_data_files where file_id=10;

FILE_NAME
--------------------------------------------------------------------------------
/opt/oracle/oradata/orcl/users02.dbf
```

下面通过 dd 命令将 499 这个数据块转储（dump）出来：

```
dd if=/opt/oracle/oradata/orcl/users02.dbf  of=a.dmp bs=8192 skip=501 count=1
```

上述命令从文件中复制出了第 501 号块，这个块就是存储这三条记录的数据块，该数据块的尾部如图 10-3 所示。

```
00001f80h: 35 38 37 35 2C 00 03 05 C4 03 4B 3B 4B 0B 62 20 ; 5875,...?K;K.b
00001f90h: 30 30 32 37 34 35 38 37 34 08 64 32 37 34 35 38 ; 002745874.d27458
00001fa0h: 37 34 2C 00 03 05 C4 03 4B 3B 4A 0B 62 20 30 30 ; 74,...?K;J.b 00
00001fb0h: 32 37 34 35 38 37 33 08 64 32 37 34 35 38 37 33 ; 2745873.d2745873
00001fc0h: 2C 00 03 05 C4 03 4B 3B 49 0B 62 20 30 30 32 37 ; ,...?K;I.b 0027
00001fd0h: 34 35 38 37 32 08 64 32 2C 01 02 02 C1 02 05 31 ; 45872.d2,...?.1
00001fe0h: 31 31 31 31 2C 01 03 02 C1 03 FF 04 62 62 62 62 ; 1111,...?  .bbbb
00001ff0h: 2C 01 03 02 C1 02 FF 04 61 61 61 61 01 06 06 F7 ; ,...?  .aaaa...□
```

图　10-3

其中，2c 01 03 02 c1 02 FF 04 61 61 61 61 就是我们插入的第一条记录（1,null,‘aaaa’），2c 是行头，这是一个标准的数据行，01 表示该行使用了 1 号 ITL 槽，03 表示这一条记录共有多少个字段（本例有 3 个字段）。02 C1 02 是第一个字段的值，这个字段是数值类型，02 表示该字段的长度，C1 02 就是十进制的 1。后面的 FF 表示第二个字段是空值，而 04 61 61 61 61 是最后一个字段 aaaa。从第三条记录来看（三条中最上面的那条记录，地址是 00001fd0h，以 2C 01 02 开头），可知每一行的第三个字节表示这一行的字段数量。为什么这一行只有两个字段呢？我们来看前面的 insert 语句，这一条记录的最后一个字段是 NULL，如果某一行的最后几个字段都是 NULL，那么 Oracle 在存储时，就会直接省略这些 NULL 的字段，以节约存储空间。

在这种情况下，如果我们执行“UPDATE TEST1 SET B=‘ABC’WHERE A=1”语句，会出现什么情况呢？这时数据块会被重组，在第一条记录中原本只有一个字节“FF”的地方插入三个字节，同时这个数据块的数据占用的顶部也上升了。

从上面的例子我们可以看出，一旦 VARCHAR 字段发生改变，就需要在数据块中额外分配空间。因此，在更新某条记录的时候，如果系统发现这个数据块已经满了，就无法在原有的数据块中存储这条记录了，那么这条记录就被迫迁移到别的数据块中。而在这条记录原本存放的位置，会被放入一个指针。在这种情况下，如果要访问这条记录，就需要在读取了这个指针后，再访问另外一个数据块，行迁移的存在降低了数据访问的性能。也许有朋友要问，为什么原来的行迁移走了还留下一个指针？不留指针不就不存在这个性能隐患吗？我们可能忘记了一点，也许这张表存在几个索引，如果我们将这一行直接迁移而不留指针，那么所有索引中和这一行相关的数据都需要进行重组，这个重组的代价远高于行迁移访问的开销。

下面我们做一个实验，首先对这张表插入一定的数据，为了证明在某些情况下使用默认为 10%的 PCTFREE 参数是多么危险，这里将表的 PCTFREE 参数设置为 10%，然后 TRUNCATE 原有的表，重新插入数据：

```
drop sequence seqt;
create sequence seqt;
alter table test1 pctfree 10;
truncate table test1;

begin
  for i in 1..1000 loop
    insert into test1 values (seqt.nextval,'abc',null);
  end loop;
end;
/
commit;
```

这时执行查询语句：

```
SQL> select a,b from test1 where a=1;

     A  B
----------
      1  abc
Execution Plan
```

```
----------------------------------------------------------
Plan hash value: 4122059633
--------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     2 |    12 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| TEST1 |     2 |    12 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("A"=1)
Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          8  consistent gets
          0  physical reads
          0  redo size
        459  bytes sent via SQL*Net to client
        400  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

由于并未创建索引，所以这个查询进行了全表扫描，产生了 8 个 CR GET。下面通过 UPDATE 字段使数据块产生行迁移：

```
update test1 set c='1234567890123456789012345678900abcdefghg'; commit;
```

这时，我们再来看数据块中发生了什么，如图 10-4 所示。

```
00001b00h: 06 20 02 00 02 80 01 F7 00 05 2C 02 03 02 C1 2E ; . ...€.?.,...?
00001b10h: 03 61 62 63 27 31 32 33 34 35 36 37 38 39 30 31 ; .abc'12345678901
00001b20h: 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 ; 2345678901234567
00001b30h: 38 39 30 61 62 63 64 65 66 67 68 67 20 02 00 02 ; 890abcdefghg ...
00001b40h: 80 01 F7 00 04 20 02 00 02 80 01 F7 00 03 20 02 ; €.?. ...€.?. .
00001b50h: 00 02 80 01 F7 00 02 20 02 00 02 80 01 F7 00 01 ; ..€.?. ...€.?.
00001b60h: 20 02 00 02 80 01 F7 00 00 20 02 00 02 80 01 F6 ;  ...€.?. ...€.?
00001b70h: 01 C4 20 02 00 02 80 01 F6 01 C3 20 02 00 02 80 ; .?...€.??...€
00001b80h: 01 F6 01 C2 20 02 00 02 80 01 F6 01 C1 20 02 00 ; .??...€.??..
00001b90h: 02 80 01 F6 01 C0 20 02 00 02 80 01 F6 01 BF 20 ; .€.??...€.??
00001ba0h: 02 00 02 80 01 F6 01 BE 20 02 00 02 80 01 F6 01 ; ...€.??...€.?
00001bb0h: BD 20 02 00 02 80 01 F6 01 BC 20 02 00 02 80 01 ; ?...€.??...€.
00001bc0h: F6 01 BB 20 02 00 02 80 01 F6 01 BA 20 02 00 02 ; ??...€.??...
00001bd0h: 80 01 F6 01 B9 20 02 00 02 80 01 F6 01 B8 20 02 ; €.??...€.??.
00001be0h: 00 02 80 01 F6 01 B7 20 02 00 02 80 01 F6 01 B6 ; ..€.??...€.??
00001bf0h: 20 02 00 02 80 01 F6 01 B5 20 02 00 02 80 01 F6 ;  ...€.??...€.?
00001c00h: 01 B4 20 02 00 02 80 01 F6 01 B3 20 02 00 02 80 ; .?...€.??...€
00001c10h: 01 F6 01 B2 2C 02 03 02 C1 15 03 61 62 63 27 31 ; .??...?.abc'1
00001c20h: 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 ; 2345678901234567
00001c30h: 38 39 30 31 32 33 34 35 36 37 38 39 30 61 62 63 ; 8901234567890abc
```

图　10-4

可以看到，部分原来的数据行变成了一些类似于 20 02 00 02 80 01 f7 00 04 这样的数据，这说明发生了行迁移，这一行已经被迁移到 RDBA=0x028001f7 的数据块中的 0x04 行中了。这时，如果访问这条数据，就需要再到 0x028001f7 做一次查询才能完成。我们再来执行刚才的 select 语句，看看发生了什么变化：

```
SQL> set autotrace on;
SQL> col b format a50 trunc
SQL> set line 132
SQL> select a,b from test1 where a=1;

         A B
---------- --------------------------------------------------
         1 abc

Execution Plan
----------------------------------------------------------
Plan hash value: 4122059633
---------------------------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     2 |    12 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| TEST1 |     2 |    12 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("A"=1)
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         17  consistent gets
          0  physical reads
          0  redo size
        459  bytes sent via SQL*Net to client
        400  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

我们看到由于行迁移，原本的 8 个 CR GET 变成了 17 个 CR GET，看样子行迁移的负面作用还是挺明显的。当然这是个特例，做了全表扫描，但是行迁移会产生负面影响是肯定的。

了解了行迁移的产生原因以及危害性后,我们再回过头来认真考虑一下 PCTFREE 这个参数。这时就会发现，这样一个小参数，里面却包含了数据库优化的大道理。如果一张表中的数据被插入后经常需要进行更新，那么必须要把 PCTFREE 参数设置得大一些，以避免行迁移的出现。如果一张表在插入数据后不做修改和删除，那么是不是就可以把 PCTFREE 参数设置得小一些，比如 5，甚至更小，这样的话这张表的每一个数据块中就可以包含更多的记录，从而减少访问这张表带来的开销。而如果有一张表，数据块的热块冲突很严重，是否也可以通过加大 PCTFREE 参

数来减少每个数据块中的记录数，从而缓解热块冲突呢？实际上，减少热块冲突更好的办法是将这张表放在 BLOCK_SIZE 较小的表空间里，不过在实际生产环境中往往难以实现，当发现热块冲突存在的时候，只能通过权宜之计来解决问题了。

在调整 PCTFREE 参数时，要注意的是，由于这个参数是可以动态调整的，因此调整 PCTFREE 参数只能对新的数据插入起作用，而对于已经填充过满的旧数据块是无法生效的。要想彻底解决行迁移的问题，必须在调整该参数后，对表进行重组，这样才能对表中的所有数据块都起作用。重组表的方法有很多，比如 ALTER TABLE ... MOVE 或者 EXP/IMP。

10.1.2　那些逝去的老参数

有几个参数随着 Oracle 数据库版本的变化逐渐逝去了，不过这些参数可能承载了很多老 DBA 的荣誉和苦泪，它们是 PCTUSED、FREELISTS 和 FREELIST GROUPS。随着 ASSM（自动段空间管理）在 Oracle 9*i* 版本的推出，现在的新数据库已经很少使用 MSSM（手工段空间管理）了。自动段空间管理和手工段空间管理最大的区别在于空闲块的管理。在自动段空间管理出现之前，Oracle 的空闲块是通过 FREELIST 机制来管理的，在创建表和索引时必须指定 PCTFREE、PCTUSED、FREELISTS、FREELIST GROUPS 等参数。数据块经过格式化后会被挂在 FREELIST 上，这样需要插入数据的前台进程就可以在 FREELIST 上找到可以插入数据库的块，并插入相关的数据。数据插入后，如果数据块中的空闲空间比例小于 PCTFREE 参数指定的比例，那么这个数据块就不能再次插入数据了，该数据块将被系统从 FREELIST 上摘除，下次插入数据时，就会插入到其他的数据块中。如果这个数据块中的某些数据被删除后，空闲空间低于 PCTUSED 参数的指标，那么该数据块将会再次被放到 FREELIST 上，这样这个数据块就可以再次插入数据了。有人可能会产生疑问，一个比较“满”的数据块，如果删除了数据，不就马上可以插入数据了吗？这时只要将其放回 FREELIST 不就行了，还用 PCTUSED 参数干什么呢？实际上，Oracle 的这个设计是十分优秀的，如果不设计 PCTUSED 这个参数，我们会碰到这样的情况：一个数据块刚刚插入数据后被系统从 FREELIST 中摘除，马上又因为删除数据被再次放入 FREELIST，那么就可能产生抖动，FREELIST 的性能就会因为频繁重组而急剧下降。

同样的道理，在一个 MSSM 管理的段中，如果 PCTFREE 和 PCTUSED 这两个参数设置得不合理，那么这种抖动现象还会出现。比如，设置 PCTFREE 为 30，PCTUSED 为 65，如果行长度较大的话，一个数据块在刚刚插入一条记录后又马上删除了两三条记录，在这种情况下，这个数据块可能会出现刚刚被系统从 FREELIST 上摘除，就马上又被再次放入 FREELIST 的现象。从 MSSM 管理空闲块的方法，可以看出设置合理的 PCTUSED 参数的重要性。如果 PCTUSED 参数设置得过高，可能会导致 FREELIST 的抖动；而如果 PCTUSED 参数设置得过低，可能又会导致数据块中存储的记录数过少，这两种情况都会影响全表扫描操作的性能。

讨论完 PCTUSED 参数，我们再来分析另外一个重要的段空间管理参数——FREELISTS。FREELISTS 用于存放可插入数据的数据块，它存在于永久数据段、临时数据段、索引段和回滚段中，一个段中可以包含三种类型的 FREELISTS：MASTER FREELISTS、PROCESS FREELISTS

和 TRANSACTION FREELISTS。

每个段中只有一个 MASTER FREELISTS，它在段创建的时候就生成了，段中高水位以下的空闲块都挂在 MASTER FREELISTS 上。当某个前台进程需要插入数据时，就可以到 MASTER FREELISTS 上去查找空闲块，查找操作从 FREELISTS 的头部开始，直到找到可以插入数据的块为止。

明眼人看到这里马上就会发现问题，FREELISTS 是一个串行的结构，如果有大量并发的会话需要插入数据，FREELISTS 就会成为瓶颈。实际上 Oracle 对此早有对策，在 Oracle 中还存在另外一种 FREELISTS，即 PROCESS FREELISTS。PROCESS FREELISTS 和 MASTER FREELISTS 构成一种主从协同机制，每个 PROCESS FREELISTS 上都链接了一组空闲块，这些空闲块是从 MASTER FREELISTS 上摘取的，前台进程要插入数据时，首先必须锁定一个 PROCESS FREELISTS，然后从这个 PROCESS FREELISTS 上查找一个空闲的数据块用于插入数据。当 PROCESS FREELISTS 上没有可用的空闲块时，会锁定 MASTER FREELISTS，从中分配一组空闲块到 PROCESS FREELISTS 上，然后再从 PROCESS FREELISTS 上分配空闲的空间。这听起来似乎很不错，Oracle 的这种设计避免了 FREELISTS 成为并行插入操作的瓶颈，不过大家要注意的是，Oracle 默认的 FREELISTS 参数值是 1，此时段头中只有一个 MASTER FREELISTS，而不存在 PROCESS FREELISTS。

不幸的是，绝大多数用户在使用 MSSM 的时候并没有设置 FREELISTS 参数，也并没有留意到这种设置带来的负面影响。这种负面影响到底有多大呢？老白遇到过的最为极端的案例是某移动公司的短信平台。这个平台每天需要插入的数据量在几千万条记录到几亿条记录之间，由于使用的是 Oracle 8*i* 数据库，因此只能采用 MSSM。我们在这个平台上进行了测试，在存在 8 个并发插入进程的情况下，如果将 FREELISTS 设置为 8，性能比 FREELISTS 设置为 1 时提高了 5.7%。在一般情况下，性能可能提升没有这么明显，但如果 FREELISTS 参数设置得不合理，造成 1%～5%的性能影响还是很有可能的。

FREELISTS 机制设计得确实比较巧妙，似乎所有的问题都已经解决了。不过细心的读者还会有一些疑问，那些已经标志为“满”的块怎么再次回到 FREELISTS 上呢？难道只要数据被删除了，块使用率低于 PCTUSED 参数值了，这个数据块就会马上被放到 FREELISTS 上吗？这么做似乎也不太稳妥，因为一旦删除操作回滚了，那么岂不是又要再次把这个块从 FREELISTS 上摘除吗？实际上，Oracle 早就为这种操作设计了算法，那就是 TRANSACTION FREELISTS。一个段默认有 16 个 TRANSACTION FREELISTS，如果某个事务要修改这个段，首先会搜索 TRANSACTION FREELISTS。如果找到了这个会话的 TRANSACTION FREELISTS，那么就继续使用；否则这个会话就会继续查找空闲的槽位。如果找不到空闲的槽位，这个会话就会试图去扩展一个新的 TRANSACTION FREELISTS，在扩展时，如果段头中正好已经没有空闲空间了，那么这个会话就只能等待其他会话提交后腾出空闲的槽位。这种等待类似于前面章节介绍过的事务槽等待，不过事务槽等待产生于数据块中，而 TRANSACTION FREELISTS 等待则存在于段头中。如果使用 MSSM 时，段头的等待较为严重，我们就应该检查一下是否是这个原因引起了等待。

假设一个会话申请到 TRANSACTION FREELISTS 后在某个数据块中删除了一些数据，那么

在这个数据块的使用率低于 PCTUSED 参数值后，该数据块就会被挂到 TRANSACTION FREELISTS 上，此时这个数据块就可以被插入数据了。不过由于这个数据块只是在本会话的 TRANSACTION FREELISTS 上，因此在事务提交之前，只有本事务可以在这个数据块中插入数据，只有等事务提交后，TRANSACTION FREELISTS 上的数据块才会被挂到 MASTER FREELISTS 的尾部，这时其他的会话才可以使用这些空闲块。

上述算法是不是很巧妙？Oracle 通过这三种 FREELISTS，很好地解决了插入数据时的性能问题，将 FREELISTS 机制产生瓶颈的可能性降到了最低。

对于 MSSM 来说，还有一个十分重要的参数 FREELIST GROUPS。一般来说，这个参数被认为是 OPS/RAC 相关的参数，实际上这种认识存在一定的片面性，不过我们还是先从这个参数在 RAC 环境下的应用说起。对于 RAC 系统来说，在 MSSM 环境下，FREELISTS 的算法和单实例环境是十分类似的，了解 CACHE FUSION 机制的读者这时就会担心了，这种机制可能在 RAC 环境下产生严重的全局缓存相关的性能问题。比如，不同的实例都对某张表进行插入，如果 Instance 1 上有一个会话在 BLOCK 10 中插入了一条记录，然后 Instance 2 上也有一个会话在 BLOCK 10 中插入了另外一条记录，接着，Instance 1 再次在 BLOCK 10 中插入记录，这样的操作如果很多的话，就会成为一场灾难，这个数据块将在两个节点中不停地来回传播，形成了 GLOBAL BUFFER BUSY。一旦这种数据块变多，那么整个 RAC 系统的性能就会急剧恶化。

事实上，Oracle 有相应的技术来避免这种悲剧的发生，FREELIST GROUPS 就是为解决这个问题而设计的。比如，目前的 RAC 环境是一个两实例的数据库。我们对某张表设置了 FREELIST GROUPS 参数为 2，那么这张表创建完毕后，在这张表的段头中会有两个块，专门用于存储两组不同的 FREELISTS（我们称之为 FREELISTS BLOCK）。这种情况下，段头数据块里只有一个 SUPER MASTER FREELISTS，上面挂载了这个段中的空闲块，而在每个 FREELISTS BLOCK 中都有一个 MASTER FREELISTS 和若干个 PROCESS FREELISTS（由参数 FREELISTS 参数确定），在这两组 FREELISTS 上分别挂了一组空闲块，当某个实例上的会话需要插入数据的时候，会通过 Instance_ID、PID、Instance 数量、FREELIST GROUPS 参数等因素来选择一组 FREELISTS，并从这组 FREELISTS 上选取空闲块。只要设置 FREELIST GROUPS 的值等于 Instance 的数量，就可以确保每个实例都有自己独立的 FREELISTS 组，由于一个数据块只能挂载在一个 FREELISTS 上，这样就确保了不同实例插入数据时，不会选择不同的数据块（FREELIST GROUPS 的值大于 Instance 数量时，一般也可以确保在大多数情况下不会出现不同实例共享相同 FREELISTS 组的情况，由于其选择算法较为复杂，这里不做详细讨论）。

这种机制很好地避免了我们所担心的问题，这是不是很棒呢？但不幸的是，我在十多年的优化工作中，还没有碰到过一套为 RAC 环境设置了 FREELIST GROUPS 参数的系统，所有的系统都是采用默认值，而 Oracle 的默认值恰恰是每个段只有一个 FREELISTS 组。更为不幸的是，一旦段创建之后，FREELIST GROUPS 就是固定的，无法动态地扩展，如果要扩展 FREELIST GROUPS，就必须对整个段进行重建。因此，在做数据字典设计的时候，开发人员应该事先为 RAC 环境设计好 FREELIST GROUPS 参数，以避免今后扩展带来的麻烦。在设置 FREELIST GROUPS 参数时，我们不仅仅要考虑到 RAC 环境，而且要为今后 RAC 节点的扩展预留足够的

FREELIST GROUPS。

看到这里，读者应该理解了 FREELIST GROUPS 在 RAC 应用中的作用，而事实上，FREELIST GROUPS 不仅仅在 RAC 环境下有用，在一些极端条件下，这个参数在单实例环境中也能发挥作用。在一个插入并发量很大的环境中（比如几十个甚至上百个并发会话会对同一张表进行插入），我们可能经常会观察到这张表的段头的等待，在这种情况下，设置 FREELIST GROUPS 大于 1 可以减少由于 FREELIST 争用而导致的段头的热块争用，多个 FREELIST GROUPS 可以将针对 FREELISTS 的并发操作分散在多个 FREELISTS BLOCK 中，从而提高并发插入的性能。

本节介绍了 FREELISTS 和 FREELIST GROUPS 参数在解决大并发量插入性能问题时的很多优点，但这并不代表我们可以滥用它们。这两个参数也有一些缺点，多个 FREELISTS 可能导致数据块中的数据填充率变低，段的高水位推进过快，存储相同数量的记录，可能会占用更多的数据块。对于表扫描较多的应用来说，我们需要慎重考虑二者的正面和负面影响，从而权衡使用这组参数。一般来说，在 OLTP 系统中，插入并发量很大的表中的数据量也很大，应用在访问这些表的数据时，通常采用索引扫描，因此加大这组参数的副作用较小。而对于 OLAP 系统，切不可为了提高数据并发加载的性能而轻易加大这组参数，因为 OLAP 系统的数据经常需要进行全表扫描，使用这组参数可能会影响全表扫描的性能。

谈了这么多，可能有些读者会说，现在主流的数据库都已经是 Oracle 10*g* 甚至 11*g* 版本了，从 Oracle 9*i* 开始就支持 ASSM 了，讨论 MSSM 是不是已经没有多大意义了呢？事实上，对于 DBA 来说，了解一些 MSSM 的技术还是有意义的，因为在日常工作中，我们很可能会碰到 8*i* 版本的数据库，也可能碰到一些 9*i* 或者 10*g* 版本的数据库，它们是从 8*i* 版本升级上来的，可能很多表空间都是 MSSM 管理模式的。另外，为了规避某些 Bug（比如，9*i* 版本的 ASSM 表空间由于对象重用导致的 ORA-600 [kcbget_xxx]，ORA-600 [kcbnew_xxx]这类 Bug），可能也需要使用 MSSM 的表空间。在某些极端的情况下，为了避开 ASSM 中 BITMAP 带来的性能问题，我们也会使用 MSSM（在绝大多数情况下，ASSM 的 BITMAP 机制在性能上优于 FREELISTS 管理模式，不过在某些特殊应用情况下，则恰恰相反）。

通过上面的讨论，大家是否已经掌握了 MSSM 下段参数的设置要点？实际上，在具体的应用环境中，可能出现的情况远比老白和大家一起探讨的要复杂得多，如何在纷繁复杂的环境中处理好这几个看似很简单，但是又很让人头痛的参数呢？这需要数据字典的设计者对 MSSM 的管理机制十分了解，另外还要足够了解自己应用的特点，这样才能做出合理的选择。最后要提醒大家的是，当你无法做出合理的判断时，做个试验可能是比较明智的选择，到底哪种方案更好，应用系统说了算。

10.1.3 减少热块冲突的方法

实际上，本节要讨论的内容在之前的章节中已陆续提及过，这里只是想通过系统的论述加深大家的印象。热块冲突是最为常见的现象，也是 DBA 讨论最多却实践最少的部分。几乎所有的系统都存在热块冲突的问题，只是严重程度不同而已，一般系统的热块冲突对系统造成的影响都

小于 5%，因此绝大多数 DBA 对此采取了容忍的态度。其实大多数的热块冲突都可以通过应用方面的优化来解决。除了修改 SQL 外，解决热块冲突最为有效的方法就是调整表的存储结构。

为了探讨这一话题，我们首先需要了解什么是热块冲突，热块冲突包含哪些形式。首先要声明的是，本节的讨论是围绕着表这个话题的，解决热块冲突的方法有很多，我们会在今后的很多话题中陆续介绍这些方法，这里主要讨论如何通过优化表的结构来减少热块冲突。

谈到热块冲突，我们首先需要了解热块冲突产生的原因，从 Oracle 对 buffer busy waits 这个等待事件的定义上可以看出一些端倪，如图 10-5 所示。

Buffer Busy Waits ID's and Meanings

Reason Code (Id)			Reason
<=8.0.6	8.1.6-9.2	>=10.1	
0	0	n/a	A block is being read
1003	100	n/a	We want to NEW the block but the block is currently being read by another session (most likely for undo).
1007	200	n/a	We want to NEW the block but someone else has is using the current copy so we have to wait for them to finish.
1010	230	n/a	Trying to get a buffer in CR/CRX mode , but a modification has started on the buffer that has not yet been completed.
1012	-	n/a	A modification is happening on a SCUR or XCUR buffer, but has not yet completed
1012 (dup.)	231	n/a	CR/CRX scan found the CURRENT block, but a modification has started on the buffer that has not yet been completed.
1013	130	n/a	Block is being read by another session and no other suitable block image was found e.g. CR version, so we wait until the read is completed. This may also occur after a buffer cache assumed deadlock. The kernel can't get a buffer in a certain amount of time and assumes a deadlock. Therefore it will read the CR version of the block. This should not have a negative impact on performance, and basically replaces a read from disk with a wait for another process to read it from disk, as the block needs to be read one way or another.
1014	110	n/a	We want the CURRENT block either shared or exclusive but the Block is being read into cache by another session, so we have to wait until their read() is completed.
1014 (duplicate)	120	n/a	We want to get the block in current mode but someone else is currently reading it into the cache. Wait for them to complete the read. This occurs during buffer lookup.
1016	210	n/a	The session wants the block in SCUR or XCUR mode. If this is a buffer exchange or the session is in discrete TX mode, the session waits for the first time and the second time escalates the block as a deadlock and so does not show up as waiting very long. In this case the statistic: "exchange deadlocks" is incremented and we yield the CPU for the "buffer deadlock" wait event.
1016 (duplicate)	220	n/a	During buffer lookup for a CURRENT copy of a buffer we have found the buffer but someone holds it in an incompatible mode so we have to wait.

图　10-5

buffer busy waits 等待事件的三个参数中的前两个是文件号和数据块号，第三个参数在 8.0～9.2 版本中都是等待原因，从 10.1 版本开始，第三个参数的含义变成了块的类别（BLOCK CLASS#）。一般来说，buffer busy waits 等待事件产生的主要原因有以下几个方面。

- ❑ 访问某个数据块时，其他会话正在将该数据块读入 DB Cache，如果 I/O 系统存在性能问题，那么会加重这种类型的等待。10*g* 版本中将这种 buffer busy waits 独立为另外一个等待事件——read by another session，这个名称更为直观。
- ❑ 访问某个数据块时，这个数据块被其他会话以不兼容的模式所持有。

产生等待的 BUFFER 可能是段头，也可能是数据块、UNDO 数据块等。段头可能由于大量的并发插入导致 FREELISTS 的等待或者 ASSM 下段头里的位图块（bmb block）等待。对待不同类型的 buffer busy waits 等待事件，我们的处理方法也是不同的。

一般来说，段头的等待主要集中在 FREELISTS 或者 BMB 上，可以通过调整 FREELISTS、FREELIST GROUPS 等参数来解决。如果等待集中在 BMB 上，那么通常只能通过使用分区表或者调整应用等手段来解决了。本节我们重点讨论如何减少表数据块上的热块冲突。

说起解决数据库热块冲突的办法，实际上有两条路：第一条路是从应用的角度减少热块冲突，

第二条路是减少热块对系统的影响。在任何一个系统中，热块冲突都是避免不了的，因此在日常优化时，我们首先应该考虑是否有可能减少热块冲突带来的影响。顺着这个思路，就可以找到一些这方面的实施手段，比如提高 DB Cache 的命中率（一般通过加大 DB Cache 的大小来实现）、使用多缓冲区技术等。在我经历过的案例中，至少有 30%是通过上述手段解决问题的。采用这些手段解决问题代价比较小，不需要花大力气去分析应用。

前面我们也讨论过，增加 INITRANS 参数可以有效避免由于事务槽等待而产生的热块冲突。加大 PCTFREE 参数的值，可以使每个数据块存储的记录数减少，从而减轻热块冲突。比加大 PCTFREE 参数值更为彻底的方法，是将表存储在 BLOCK_SIZE 更小的表空间上。

如果数据块本身存在热点该怎么处理呢？这个问题回答起来有点麻烦，因为既然是数据，就必然存在多样性，所以不存在能够解决问题的“灵丹妙药”。除了刚才提到的方法可以缓解热块冲突外，Oracle 还提供了一系列的方案，最为典型的是 HASH 分区表和 HASH 簇表。

HASH 分区表是解决热块冲突的一种较为常用的办法，对于表数据量较大的情况，可以考虑采用 HASH 分区表。比如，一张表的主键是通过 SEQUENCE 产生的，那么在没有使用 HASH 分区表的情况下，同一个时间点产生的记录存储在同一个数据块中的可能性很大。而这些数据随后又被其他应用使用，这样产生热块的可能性就很大了。但如果将这张表根据主键设计成 HASH 分区表，那么同一个时刻产生的记录就被散列算法分布到不同的表分区中去了，访问这些数据的时候就可以从多个数据块中读取，从而缓解了热块冲突。

既然 HASH 分区表这么强，肯定有些朋友会蠢蠢欲动，干脆把存在热块争用的表都设计成 HASH 分区表好了。实际上，任何技术都有其两面性，HASH 分区表解决了热块冲突的问题，但是又带来了另外一个问题。如果应用总是通过主键来访问这张表的数据，那么这种方式确实是最好的。但是如果还有大量的应用需要根据主键进行范围扫描，或者按照记录的生成日期进行范围扫描，那么 HASH 分区表的弱点就显现出来了。原本放在同一个数据块中的数据被散列算法分散开了，这同时意味着我们对这些数据做范围扫描的时候需要扫描更多的数据块。这就是 HASH 分区表的弱点，它增加了范围扫描的成本。

在实际的生产环境中，我们可能不总是那么幸运，肯定会碰到两方面的需求：一方面必须解决热块冲突，另外一方面可能还存在一些应用要对这些数据做范围扫描。在这种情况下，必须进行综合的评估，确定到底哪种需求才是主要需求。如果优化范围扫描对系统更有利，那么就必须放弃 HASH 分区；而如果解决热块冲突更为重要，那么就必须牺牲范围扫描。实际上，Oracle 就是这样的。任何技术都是矛盾的，有缺陷的，否则我们只需要记住一些准则，就可以成为大师了。成长为 Oracle 大师并非易事，因为在绝大多数情况下，没有一成不变的准则。

HASH 簇表（HASH cluster table）不是分区表，但是它和普通表（也就是术语所说的堆表，heap table）的存储方式是不同的。尽管 HASH 簇表是簇表的一种，但其组织方式是根据散列值来进行的，不同的数据块中存储不同散列值的行。这样一来，即使是同时插入的两条记录，也有可能因为某个字段的散列值不同而存储在不同的数据块中，从而最大限度地减少热块冲突。

HASH 簇表主要适用于以等于条件对表进行访问，且很少根据 CLUSTER 键值进行范围扫描的情况。HASH 簇表除了能够优化热块冲突外，对于改善大表的 SELECT 操作也有一定的作用。

使用下面的语句可以创建一个 HASH 簇表：

```
SQL> CREATE CLUSTER off_clu
  2    ( country    VARCHAR2(2),
  3      postcode   VARCHAR2(10) )
  4      SIZE       350
  5      TABLESPACE case_large_data
  6      HASHKEYS   6000;
SQL> CREATE TABLE office
  2    ( office_cd   NUMBER(3),
  3      cost_ctr    NUMBER(3),
  4      country     VARCHAR2(2),
  5      postcode    VARCHAR2(10) )
  6    CLUSTER off_clu(country, postcode);
```

创建 HASH 簇表时的主要参数包括：

- HASHKEYS，KEY 的数量。
- HASH IS，使用客户自定义的散列函数。
- SIZE，预留的空间，用以存储相同散列值的不同 HASHKEY 值。如果 HASHKEY 的长度比较大，建议设置较大的值。

除了 HASH 分区表和 HASH 簇表外，仍然有很多种方法可以用来减少热块冲突。加大 PCTFREE 值就是一种常用且简单的方法。PCTFREE 参数值增加后，平均每个数据块中的记录数就会减少，从而减少产生热块冲突的机会。

还有一种办法是使用较小的块。从 9*i* 版本开始，Oracle 提供了一种新技术，用户表空间可以设置独立的块大小，而不需要和数据库的默认值相同，这种技术为解决热块冲突提供了另外一种选择。不过在实际应用环境中，我们很少能见到使用了这一技术的系统，这是因为开发厂商往往不了解这种技术，因此也很少设计这样的系统。

至此，关于表及相关存储参数的讨论就告一段落了，但在后续章节中我们还会继续讨论这个永恒的话题。从下一节开始，老白将通过一个实际的优化案例来重申表结构及参数设置的重要性。

10.2　从数据块结构看目前主流容灾技术

谈到容灾平台，很多 DBA 总是觉得这离自己很远，也许自己一辈子也看不到容灾系统的启用。有一次在四川碰到几个朋友，谈起了容灾平台，我说真正能启用的恐怕不到一半，而其中一位以前在某省邮储从事容灾工作的朋友则表示难以置信，他觉得当年自己开发容灾系统时是很认真的，而且有专人管理，定期演练时必须确保能够立马切换。也许我碰到的客户都不是金融行业的，容灾建设比较马虎吧。

正巧此时一个金融行业的客户打来求助电话，他们接到上级部门的通知，要进行容灾演练，而在切换容灾系统时，却发现系统中一套 10*g* 版本的 Oracle 数据库无法启动。

后来问题解决了，其实很简单，他们的容灾环境采用的是 IBM 的 SVC 复制，但是只复制了

数据库相关的文件，并未同步操作系统的系统目录中的文件，而且 host 文件的配置也是错误的，所以才导致数据库无法启动。这个问题虽然不大，却暴露出另一个问题，就是容灾平台疏于管理，当主生产环境变更时，没能及时变更容灾环境。在这些年的工作中，老白亲自经历过三次容灾切换，其中两次成功，一次失败。成功的两次都是切换到本地容灾系统，而失败的那一次则是切换到远程容灾系统。

先说说成功的那两次吧。当时客户要为系统添加 RAC 第三节点，由于存储容量不足，无法满足第三节点 UNDO 表空间和归档目录的需求，因此决定扩充一个由 16 块硬盘组成的扩展柜。由于集成商经验不足，新扩展的盘柜和磁盘的微码与原系统不兼容，扩容工作很不顺利。原定于周五晚上添加 RAC 节点，周四必须完成磁盘扩容工作，但是从周三就开始的盘阵扩容进行了一天半还是毫无进展。于是 IBM 的售后服务人员也加入了此项工作，经过诊断，他们建议现场插拔一下磁盘。在反复了几次后，磁盘突然整个锁死，系统无法正常工作了，而这时正好是下午 5 点多，系统业务最高峰的时段。

经过我和客户技术主管的讨论，我们决定马上切换到容灾系统。当时此系统有两套容灾系统，本地容灾采用 DATAGUARD 技术，远程容灾采用类 DATAGUARD 技术将归档日志压缩后传输到远程进行注册和应用。远程容灾系统在离故障点 100 多公里以外的城市，配备了两台 IBM P561 服务器，而本地容灾系统只是利用剩余设备搭建的一个备份平台，其目的并不是为了容灾，而是为了减轻备份对生产系统的负载。它只包含一台 IBM P561 服务器，且仅配备了 48 GB 的内存，此配置与生产环境两台配备了 72 GB 内存的 P570 系统相比，处理能力明显不足。按照常理，在这样的情况下，切换到异地容灾系统是比较合理的，但是经过仔细分析，当时并不具备这样的条件，因为只有主生产系统在异地建立了容灾系统，而其余十多个外围系统并未建立相应的异地容灾系统。一旦将主生产系统切换，那么就需要更改上千个配置项才能够完成全业务的切换，而这项工作并无现成的预案和操作脚本，仅凭人工操作，很难保证系统正常切换。另外还有一个更大的问题，在远程容灾机房中，仅配备了几个初级的系统管理人员，并未配备专职 DBA 等维护人员，一旦进行切换，大家只能在远程操作系统，很难确保切换过程不出问题。

如果切换到本地容灾系统，只需要将 DATAGUARD 服务器的 IP 改为生产环境的 IP，然后激活 DATAGUARD 即可完成，而且维护人员还可以在本地进行操作。不过最大的问题是本地的硬件配置明显不足以支撑生产环境 50%的业务。于是我建议将生产环境上的部分内存插到容灾系统的服务器上。这个主意似乎很不错，但是由于生产环境采用的是 P570 服务器，而本地容灾系统采用的是 P561 服务器，大部分从 P570 上拔下的内存条并不能在 P561 上使用。于是只能又在备件库房凑了凑，才凑够了 42 GB 内存，将容灾服务器的内存扩充到了 90 GB。在随后重启容灾服务器的过程中，又碰到了网卡不能自动激活的问题，总之经过一系列的处理，终于在一个半小时后恢复了系统的运行。虽然本次切换最终成功了，不过由于业务高峰期系统停止了近 100 分钟，大量业务被积压，并影响了货物通关，最终导致了惊人的损失。

在这次事故中，投入大量资金的异地容灾系统并未发挥应有的作用，反而是不在规划中的 DBA 的无心之作解救了这次危机。这个案例也暴露出了此容灾平台建设中的问题，这些问题其实是普遍存在的。

第二个案例差不多是 10 年前的事情了，这个客户采用的容灾技术和第一个案例不同，是采用存储级复制技术，容灾平台配备了 70%的处理能力。那次老白正好在客户现场进行另外一套系统的优化。由于生产系统的存储突然掉电，大量的数据文件损坏了。当时客户还很淡定，因为他们投入巨资建设了先进的容灾系统，而且这套容灾系统在几个月前还做过切换演练。但是容灾系统启动时，其中的一台数据库服务器启动不了，从错误信息来看，是由 ORA-704 和 ORA-600 错误引起的。当值 DBA 忙碌了几个小时还是没办法解决问题，突然想到这边还有一个做优化的 DBA，于是就请我帮他解决。现在看来，那个问题其实并不难，主要是由于 UNDO 表空间存在坏块才导致数据库无法正常启动，通过_offline_rollback_segments 和_allow_resetlogs_corruption 参数，再辅以 BBED，强制打开数据库就有可能解决问题。不过那时老白的技术还比较差，整整折腾了一天也没能解决。由于客户建设了容灾系统，就没有再建设备份系统，因此这个系统没有可用的备份集，最终老白只能祭出 dul 这个法宝，导出了数据并重建了整个数据库。虽然最终大多数数据被恢复了，不过整个业务却中断了 5 天之久。

和 10 年前不同，现在的容灾系统建设已经成为一种主流，一般来说，核心的业务系统都会建设容灾平台。不过和 10 年前类似的是，现在的 IT 部门决策者还是不了解容灾技术的本质，因此在选择容灾平台的策略上，仍存在很多误区。如果不能真正从原理上理解容灾技术的本质，那么就无法保证容灾切换失败的悲剧不再重演。

目前的主流容灾技术包括下面几种：

- 存储同步复制技术；
- 存储异步复制技术（包括各种类型的存储复制以及卷复制）；
- Oracle DATAGUARD；
- 逻辑复制技术（比如 STREAMS、GOLDENGATE、DSG 等）；
- 应用级数据复制技术。

存储同步复制技术也就是平常所说的镜像（mirror）技术，一个写 I/O 必须在本地和容灾系统上同时完成写操作才算操作完成，任何一个写错误都将导致整个写 I/O 失败。这是一种十分严格的同步机制，因此能够确保容灾平台的数据随时都是可用的。这项技术是十分成熟的，甚至 Oracle 都专门提供了一套解决方案——RAC on Extended Distance Clusters，就是一套远程的 RAC 系统，两个节点不共享一个磁盘阵列，而是共享两个互为镜像的磁盘阵列，每个读写操作都会发送到两个磁盘阵列上。一旦某个地方出现故障，那么另外一个地方的系统还可以独立工作。这是一种兼顾 RAC 高可用和异地容灾的解决方案。虽然目前在国内还缺乏成功案例，但是其技术成熟度已得到验证。不过采用镜像技术的系统，一旦某个存储出现故障，必须尽快隔离，否则就会影响系统的运行，因为一个 I/O 操作必须同时成功才能够完成。

存储异步复制技术目前在容灾系统上的应用十分广泛，这种方式既提供了易于维护的高效容灾复制，又避免了备用存储故障导致生产系统无法正常工作的问题。但是如果深入研究这种容灾技术的本质，我们还是会发现其中潜在的风险。为什么会碰到这样的案例呢？这个问题在 10 年前曾使老白十分困惑，不过随着对 Oracle 内部结构的认识，特别是对数据块结构的认识，老白终于明白了，导致当年那个问题的最根本原因就是“块断裂”，其产生的根源在于 Oracle 的数据块

和操作系统的数据块大小不一致。一般的 UNIX 系统，其磁盘块的大小是 512 B，而 Oracle 的数据块大小是 8 KB，也就是说，一个 Oracle 数据块包含了若干个磁盘块。从 Oracle 数据块的定义上来看，虽然 Oracle 的数据块分配的磁盘空间是连续的，但由于底层存储条带技术，系统不能确保 Oracle 数据块在磁盘上是连续分布的，甚至一个 Oracle 数据块有可能被存储在多个磁盘中。由于 Oracle 数据块和操作系统磁盘块之间的不同，就产生了一种可能性：一个 Oracle 数据块被操作系统分成了多个 I/O 写盘，这些 I/O 之间的时间点是不同的，因此在某个瞬间，远程容灾系统上数据中可能包含了一个数据块的某个部分的变更，但是缺少了其他部分的变更，这就导致该数据块中的各个组成部分是不一致的。通过块头和块尾以及校验和的比较，就能发现数据块处于不一致状态。这种情况的表象就是 Oracle 读到了坏块，也就是我们常说的块断裂现象。这就是为什么使用存储异步复制技术的容灾系统，在数据库打开后经常会发现坏块的原因。如果坏块正好出现在 UNDO 表空间中，或者出现在一些关键的系统数据字典表中，那么在打开这个数据库时就可能会出现问题。

对于一般的系统，块断裂发生的几率并不是太高，但是对于数据变更十分频繁的 7 × 24 系统来说，出现的几率会大得多。之所以那个客户在容灾演练时没有发现问题，主要是因为客户在此期间通常会停掉业务，这样数据库就处于相对的静态，出现问题的几率很小，而实际生产系统故障需要切换时往往是业务高峰，出现块断裂的可能性几乎是 100%。

这类容灾技术有很多，主要的方案提供厂商包括 EMC、赛门铁克（赛门铁克采用卷复制技术）、飞康、HP、IBM 等。原厂的工程师总会宣称他们的算法是经过严格认证的，能够确保容灾系统上的数据完全复制自生产系统上某一瞬间的数据，并且在时间点上肯定是一致的，因此不存在不可用的问题。这个说法看似很有道理，不过仔细考虑一下，好像还是存在漏洞。时间上的一致性，就能确保数据库的一致性吗？要解决这个问题，就需要认真研究数据库一致性的条件。我们假定有一个一致性的存储快照（snap），在这个快照上，没有任何一个数据块是断裂的，那么我们认为这个快照是一致性的；反之这个快照是非一致性的，在这种情况下打开数据库，可能就会出现坏块。如果坏块出现在 SYSTEM 表空间或者 UNDO 表空间中，数据库甚至可能无法打开。即使某个快照是一致的，数据库就一定能打开吗？在进行数据库介质恢复时，如果在某个 SCN 停止了恢复，系统提示某些文件还处于不一致状态，此时打开数据库会出现故障，这种情况下，即使 RECOVERPOINT 和 SCN 是完全一致的，数据库中的某个数据文件也可能存在问题（可以通过 V$DATAFILE_HEADER 的 FUZZY 字段来判断是否有文件的 FUZZY 是“Y”，这个视图的数据是从文件头中读取的），这时打开数据库，某些数据文件可能就会出现坏块，如果是 SYSTEM 或者 UNDO 表空间的文件中存在坏块，那么数据库打开后，有可能还是会因为 ORA-600 错误而无法正常工作。

从这一点，老白想到了几年前和几个赛门铁克的工程师讨论他们的基于卷复制的容灾平台。当时的生产库是一套 10*g* RAC 和一个逻辑 STANDBY 实现的报表系统，需求是当容灾发生时，系统可以顺利切换并且报表系统的逻辑 STANDBY 还能继续工作，也就是说容灾系统上的生产库和报表系统之间的逻辑 STANDBY 必须能够继续保持关系，这就要求切换的时候两个系统还要保持 SCN 的一致性。起初，赛门铁克的工程师也认为他们的产品是无懈可击的，后来经过测试发

现，切换时经常会有坏块出现。虽然如此，他们总是能够从中找到一个时间点，在这个时间点上做切换总是可以完全恢复。实际上，这个案例从侧面印证了老白的观点，由于 SCN 在时间上的不连续性，导致一部分时间点的数据是一致的，而另一部分时间点的数据是不一致的。因此采用类似技术的容灾系统，不论其技术如何先进，都存在相同的隐患。

既然存储复制容灾容易出现块断裂，那么它是不是就没有用武之地了呢？这是不能一概而论的。随着存储复制容灾技术的不断改进，现在的技术又有了新的发展。为了提高存储容灾系统启用时的可用性，可以在每天业务量不是很高的时候，对主要表空间设置 BEGIN BACKUP 状态，然后做一个快照，再设置为 END BACKUP。一旦激活容灾系统时出现了 ORA-704 或者 ORA-600 这样的错误，就可以从那个时间点通过归档日志 RECOVER DATABASE 恢复数据，从而获得一个可用的数据库。

在官方文档 *Supported Backup, Restore and Recovery Operations using Third Party Snapshot Technologies* [ID 604683.1]中，Oracle 提出了一个不需要 BEGIN BACKUP/END BACKUP 也能实现一致性复制的设想。这篇文章往往令很多存储厂商认为，自己的存储产品是不使用 BEGIN BACKUP/END BACKUP 就总能够在容灾系统正常打开数据库的。因为 Oracle 的官方文档中有这么一句：Oracle will officially support the following operations assuming that the third party snapshot technology can meet the prerequisites listed in the next 2 sections.（Oracle 支持下列操作，只要第三方快照技术符合下面两节里列出的前提条件。）

不过如果你继续阅读下面的前提条件：

第三方快照前提条件

第三方厂商需要保证其快照符合如下要求：

(1) 与 Oracle 推荐的上述恢复操作完好集成；

(2) 数据库在快照点上一致性宕掉；

(3) 快照中对每个文件的写顺序加以保留。

就会发现这种场景对快照的要求十分苛刻。第一点和第三点比较容易实现，目前 EMC 的 SRDF、HDS 等产品都支持根据写顺序的数据复制。而第二点“数据库在快照点上一致性宕掉”是十分苛刻的。在真正的容灾环境下，有可能总是能幸运碰到这种情况吗？很多客户在测试类似的容灾平台的时候，总是通过突然杀掉生产库的实例来验证容灾是否起效，这么测试肯定是百分百的成功，因为这种情况对于容灾库而言，就像是数据库实例宕掉后重启一样。如果你尝试在生产库直接关掉存储（包括备用电池），这种情况下，你的容灾库是否还能很幸运地启动呢？我想如果这样，你真的十分幸运。

Oracle DATAGUARD 是一种适合本地容灾的解决方案，由于 DATAGUARD 是基于 Oracle 数据块的复制技术，它不会造成块断裂，因此很适合本地容灾系统使用。DATAGUARD 的缺点是，如果采用 LGWR 同步模式，一旦 DATAGUARD 出现故障，将导致生产库也无法使用；而如果采用 LGWR 异步模式或者 ARCH 传输模式，一旦主库故障，丢失的数据量可能会多于存储复制技术的丢失量。

除了上述几种常见的数据库容灾技术外，近年来，又出现了以 Oracle GOLDENGATE、DSG、

Oracle Streams 为代表的逻辑复制解决方案。这些解决方案基本上都是通过对 Oracle 日志进行挖掘，将逻辑变化记录捕获后传输到目标端，转换为 SQL 语句，并应用到目标数据库，从而实现数据同步。目前已经有很多用户使用这些产品建立了自己的容灾平台。这种容灾手段避免了数据块断裂的问题，在某些场合下是适用的。不过这种基于逻辑复制的技术存在一个很大的缺点，就是无法有效控制数据复制的延时，对源端的系统依赖较大，而且对大批量数据维护工作有较严格的限制。而这些逻辑复制产品，对于大事务的控制都存在一定的缺陷，比如，生产环境做了一个 UPDATE 操作，修改了 2000 万条记录，这个操作在生产环境中可能 10 分钟就完成了，而在目标端，该操作就变成了 2000 万个独立的 UPDATE 语句，执行完成这些 UPDATE 操作，可能需要几个小时，甚至更长的时间，这样就会造成复制的延迟。如果这时生产系统出现故障，那么恢复业务可能需要较长的时间，甚至有可能出现一些更为严重的问题。比如对于 GOLDENGATE 来说，其捕获进程在某个事务没有提交时不会捕获数据，只有当事务提交时才会捕获，如果某个事务执行的时间比较长，那么捕获进程会等待该事务提交，这样就会产生较大的延时，如果这时系统出现故障，就会出现大量没有来得及捕获的数据，这些数据可能会彻底丢失。

针对大事务，GOLDENGATE 进行了一定的优化，比如对于 INSERT 操作，GOLDENGATE 会自动合并类似的语句，采用 BULK INSERT 的方式处理，这种方式已经被证明是十分有效的，对于批量插入操作的复制效果很好。不过对于 UPDATE 和 DELETE 操作，上述处理方式并没有实现。系统是复杂多变的，实际环境并不总是以我们个人的意愿而改变。

只有了解了主要复制技术的基本实现原理，我们才能在设计自己的容灾平台时选用正确的方案。

逻辑复制技术对于数据量不大，很少有大事务的系统是有效的，但如果将其用于容灾系统，就需要加强对系统运行状况的监控。通过心跳表监控可以及时发现延迟增大问题。在一个逻辑复制环境中，目标系统出现性能问题，或者目标系统中某个不合理的查询，都有可能让复制延时变得很大（老白以前甚至还碰到过由于没有及时在目标环境中进行表分析而导致复制进程采用了错误的执行计划，从而使得复制延时变大）。另外，在逻辑复制环境中，应尽可能保证目标服务器的性能，从而避免复制延时。

DATAGUARD 适用于本地容灾，对于一些数据变化比较大的系统，传输大量的归档日志需要很高的网络带宽。

存储级别数据复制技术由于传输的仅仅是存储数据的变化，其传输数据量小于 DATAGUARD，因此很适合在广域网上进行数据复制。这种复制模式很适用于异地容灾平台。

对于不同的应用，不同的 SLA，选择合适的容灾技术十分关键。在本地建立 DATAGUARD 本地容灾系统，远程使用基于存储复制技术的远程容灾系统，也许是不错的选择。

10.3　案例——简单任务

本节内容摘自老白的日记。

11 月 24 日　令人意外的开始

今天是我第一天到用户现场，在香格里拉酒店和老方会合。老方在原厂，出差必须享受五星级标准，而我觉得 400 多元钱的酒店有点奢侈，就住在了旁边的迎宾馆。迎宾馆是原来的市委招待所，后来改造成了一个三星级酒店，房间虽然有点旧，但是十分干净。院子很大，里面非常安静。最关键的是，一个市政府的朋友给我打了个很不错的折扣，不到 200 元的价格是我选择这里的主要原因。

客户在开发区，从市区打车过去要 40 多分钟，在车上老方给我介绍了这个项目。老方是这个项目的负责人，前期也担任了售前的工作。听老方介绍，这是一个短信平台的优化项目，用户的优化目标是将系统整体的处理能力提升 25%以上，不过并没有对 CPU 资源、I/O 等提出明确的要求。乍一听，这个优化指标好像并不高，老方也觉得这是个很简单的任务，只要花点心思优化几个 SQL，应该不难达到。在出租车上的这段时间，我们俩都很轻松，老方甚至考虑到不要过早结束项目的问题，因为客户出了一个不错的价格，如果我们过去两三下就解决了问题，客户或许会觉得钱花得有点冤枉。

上午是项目启动会，甲方的项目经理姓余，是个 80 后的美女，高挑的身材，说话甜甜的。从甲方对项目的期望来看，确实像老方所说，不算太高，相比我们以前做过的项目要简单一些。这是一个短信平台的系统，在平时是没有任何问题的，只有在中秋节和春节期间会出现性能问题，导致短信话单无法正常入库，严重的时候，为了确保短信系统能够正常运作，会出现应用软件被迫丢弃话单的情况。对于运营商来说，丢弃话单就意味着话费的流失，每年由此导致的话费流失可达数百万之多。这还不是最大的问题，由于丢失的短信话单有可能是某个 SP 的业务短信，这就会导致 SP 业务收入方面的损失。为了确保业务高峰期不丢失短信话单，甲方希望本次优化能够将目前每秒最大入库短信量从 1200 条左右提高到 1500 条以上。对于其他方面的优化，能实现多少就算多少，只要事务平均响应时间能够提升 20%以上，项目就可以验收了。

今天参加开工会的除了甲方外，还有开发商的技术人员，他们对这次优化的态度十分友好，也希望我们能够对系统的性能提出一些有效的建议，以便于日后改进。另外他们还表示，只要我们的优化建议可靠，他们一定会在 20 个工作日内完成修改工作。

这个会只开了个把小时，就在友好的气氛中结束了。甲方的小余帮我们在旁边的 ADC 项目组办公室里找到了两个空座位，然后就是申请网禁，申请工作账号等。运营商在安全方面管理得比较严，我们填写了数份表格，复印了身份证，甚至在安全责任书上按了指印，终于可以连到服务器了。看看时间已经是中午 11:50 了，于是我和老方锁了电脑屏幕，到外面去找吃饭的地方。

老方对这个地方也不熟悉，我们俩转了半天，终于发现了一条比较热闹的街道，街道两边有几家饭店和一个不大不小的超市。老方是山西人，比较喜欢面食，于是我们选定了一家规模比较大的饺子馆。虽然是在西南，但是这家饺子馆还颇有北方的风味，两个人吃得都比较满意。回来的路上，我顺便在超市买了一个茶杯。我这个人喜欢喝茶，不过出差总是忘记带杯子，于是每到一个地方，总是先找超市买茶杯。还好我喜欢的是绿茶，随便找个茶杯就可以泡。

回到办公室的时候，已经快两点了。客户下午两点上班，办公室里不少人还躺在简易床上睡

午觉。我先把杯子洗了洗，泡上一杯茶，然后登录到系统看了看。这个短信平台分为两个区，分别负责半个城市的业务。这两个区根据机房的地点被命名为 A 系统平台和 B 系统平台，服务器是 IBM 的 P 系列，操作系统是 AIX 5.3。A 平台是早期建设的，数据库是 8.1.7.4 版本；B 平台是前几年扩容的，数据库版本是 9.2.0.6。

我首先查看了两套系统的 STATSPACK，都是新装的，并开了自动采样，估计这还是老方上个月来这里做售前的时候装的。接着，我在 A 平台上生成了一份今天上午 9:00 ~ 10:00 的 STATSPACK 报告，报告显示每秒逻辑读和物理读的数量都很小，逻辑读只有五六千，平均每秒的事务数也只有 0.93。想起刚才小余说过，这套系统平时的负载很小，CPU 使用率只有 20%左右，I/O 也很闲。确实如她所说，现在是下午两点多，CPU 的使用率只有 10%多一点。我继续往下看 STATSPACK 报告，并没有发现什么开销较高的 SQL，排名第一位的 SQL 是一条 INSERT 语句，它向一张 SM_HISTABLE 表插入一条记录，根据表名判断，这应该就是那张短信话单表。这条语句在一个小时内执行了 59 万次，平均算下来每秒不到 200 条，每次执行产生的逻辑读数量为 14.2。在浏览了一遍 STATSPACK 报告后，除了部分 SQL 没有使用绑定变量，硬分析比例略高之外，我并没有发现什么明显的问题。因为没有采集到业务高峰的数据，所以没有明显的问题也很正常，不过我突然意识到一个较为严重的问题。在我浏览 TOP SQL 这一节时，看到的单次执行开销最大的 SQL 每次执行产生的逻辑读操作只有不到 2000 个。这说明系统中并没有很多高开销的 SQL，这样一来，早上我和老方商定的优化几个高开销 SQL 的计划就很难实施了。

我急忙把老方叫到门外，把刚才的发现告诉了他。老方也觉得有点意外，由于这次客户的优化目标并不高，他认为达到优化目标没有任何问题，因此对于这套系统，他并没有像以往那样在售前阶段进行详细的分析。

我发现这套系统其实十分简单，除了那条开销最大的插入语句，只有少量的 SELECT 语句，而且大多数 SELECT 语句都是根据主键 MSG_ID 访问的，只有少量通过 PHONE_NO 查询。对这样一套系统进行优化，想从 SQL 入手，估计难度较大，我们必须另辟蹊径。

如果不从 SQL 入手，我们该从哪里开始呢？这么简单的应用，想要通过操作系统和数据库参数方面的调整提升 20% ~ 30%的性能，是不太现实的。调整应用的架构可能是最佳选择，不过这样的话，对应用调整过大，开发商估计也会有很大的阻力。

讨论了半天，我和老方也没想出什么好的办法，于是我建议还是先找开发人员了解一下应用的情况再说。从目前的情况看，只能先搞明白应用的关键瓶颈，再去想其他办法了。

经过了解，我大体明白了这个系统的关键问题。这是一个短信平台中的后台系统，用于存储和管理短信话单，来自短信平台的话单首先会存储在内存缓冲中，然后被后台进程批量写入数据库。每个平台都配置了 8 个并发的写入进程，用于短信话单的写入操作，每次写入的批量为 800 条。开发人员也曾尝试过增加写入进程和增加写入批量的数量，不过效果都不明显。SM_HISTABLE 是一张分区表，首先他们设计了每个月生成一张话单表，表名为 SM_HISTABLEyymm，每张表又按照每天一个分区，分为多个分区，分区主键是 CREATED_DATE。每条话单都有唯一的主键 MSG_ID，话单插入后，后续处理主要根据 MSG_ID 来查询话单数据，在更新数据时，一般都采用 ROWID 条件进行，这方面也不会有性能问题。

从这种简单的应用系统来看，我们能够进行优化的点并不多。看样子我们需要认真地考虑一下这个特殊的应用了，想想如何来完成这个现在看来并不简单的“简单任务”。

11 月 25 日　解决之道

今天一早，老方就回北京了。昨天晚上我和老方一直在讨论这个项目，从采集到的信息来看，只能通过调整操作系统、数据库的参数，以及调整表的一些参数来达到提升系统总体性能的目标。而对于系统性能来说，最为关键的也就是大量的话单插入操作。目前话单插入是由 8 个并发进程完成的，每次的插入批量是 800 条记录。

虽然实际情况和预先估计有出入，但也只能硬着头皮上了。今天我会主要分析在哪些环节可以进行优化，以及每个环节可能的性能提升比例，从而为制定优化计划提供参考。另外，采集性能基线的工作也在同时进行，由于这个项目的特殊性，我们无法采集到去年春节期间的数据。对这种系统来说，进行非业务高峰期的性能数据比较，意义并不是很大。通过昨天的分析可以看出，平时每天的话单量在 2000 万条左右，而在中秋节和除夕，话单量可能达到 1.8 ~ 2 亿。按照繁忙时集中系数为 0.1 计算，平时每天最忙的 1 个小时内，产生的话单量为 200 万条，折算到每秒，产生的话单量约为 555 条，而业务高峰期的量可能达到 5550 条。平均分配到两个系统中，每个系统每秒要处理近 3000 条话单才能确保系统不出问题。由于系统中有内存缓冲区，可以进行一定程度的平滑高峰处理，因此开发商认为，只要能够达到每秒 2000 条话单以上的处理能力，就可以保证不丢失话单，而目前的测试结果显示，系统处理能力仅能达到 1500 条。这就意味着，通过优化使性能提升 50%，才可以确保万无一失，而提升 25%以上，只能基本达到最低要求。仅仅通过调整系统参数和表的存储参数来实现这个目标，确实具有一定的挑战性。

上午我分析了系统的主要等待事件，以及各个缓冲区的情况。

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.98       Redo NoWait %:  100.00
            Buffer  Hit   %:   99.69    In-memory Sort %:   99.97
            Library Hit   %:   99.79        Soft Parse %:   88.82
         Execute to Parse %:   98.61         Latch Hit %:   99.98
Parse CPU to Parse Elapsd %:   93.32     % Non-Parse CPU:   98.51

 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:   93.70   95.50
    % SQL with executions>1:   35.74   34.34
  % Memory for SQL w/exec>1:   29.93   21.30

Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                              Wait     % Total
Event                                               Waits  Time (cs)   Wt Time
-------------------------------------------- ------------ ------------ -------
db file parallel write                              2,922       36,456   47.99
db file sequential read                            16,401       15,874   20.90
log file parallel write                             8,824       10,812   14.23
log file sync                                       2,907        5,169    6.80
```

```
direct path read                                     17,046        3,639    4.79
          -------------------------------------------------------------
```

从主要缓冲池的指标来看，没有明显的问题。而从主要等待事件上看，大多都集中在和 I/O 相关的项目上，这说明 I/O 系统的写性能指标并不好。另外，REDO LOG 相关的等待事件占整个等待事件的 21%，REDO LOG 也存在优化的可能。

从 STATSPACK 报告上看，每个小时的 redo buffer allocation retries 为 483 次，目前的 LOG BUFFER 为 1 MB，加大 LOG BUFFER 可以减少这方面的等待。目前系统共有 16 个日志组，每个日志组一个文件，每个文件 60 MB。从日志切换情况来看，今天业务高峰期平均 1 分钟多就要切换一次日志，由于无法查看去年中秋节期间的数据，因此我只能推算，如果业务量加大 10 倍，那么每隔五六秒就会产生一次日志切换，这样对系统的性能会产生较大的影响。因此，加大 LOG BUFFER，增加 REDO LOG 文件的大小，使性能提升 10%～15%还是很有可能的。

从今天的 STATSPACK 报告中，我们没有看到明显的 buffer busy waits 等待，一个小时只有几百个而已。通过测算，现在每个系统每秒的话单量不到 300 条，而每个插入的批次是 800 条，因此多个插入进程同时插入数据的机会较少，热块冲突不会很明显，而如果业务量增加 10 倍，那么热块冲突可能会变得很严重。目前 A、B 两套系统的数据库版本分别是 8.1.7.4 和 9.2.0.6，都没有使用 ASSM。通过检查发现，FREELISTS 参数都是默认值 1，这对于并发插入操作会有较大的影响。加大 FREELISTS 参数，不仅可以减少并发插入时分配空块的等待，而且不同的插入进程会使用不同的 FREELISTS，不会往同一个数据块中插入数据，也能避免不同的插入进程之间在业务高峰期间可能出现的热块争用。根据以往的经验，如果 FREELISTS 参数设置得合理，减少了这方面争用，可能带来的性能提升能够达到 5%～10%。

目前核心表 SM_HISTABLE 会每天生成一张独立的表，每张表都是分区表，表中设置了一个专门的分区字段 ID_HINT 进行范围分区，这个字段的值是循环使用的，到达 10000000 后自动回绕为 1，分区脚本如代码清单 10-1 所示。

10

代码清单 10-1

```
PARTITION BY RANGE (ID_HINT)
)
  PARTITION PART_01 VALUES LESS THAN (1000000)
    NOLOGGING
    TABLESPACE CQYDSMSC_CENTER1
    PCTUSED    40
    PCTFREE    10
    INITRANS   1
    MAXTRANS   255
    STORAGE    (
                INITIAL          504K
                MINEXTENTS       1
                MAXEXTENTS       2147483645
                FREELISTS        1
                FREELIST GROUPS  1
                BUFFER_POOL      DEFAULT
               )
)
```

这种分区模式对于减少热块冲突帮助不大，尽管在调整了 FREELISTS 参数后，热块冲突的严重程度会有所下降，但对性能的影响不会特别大。但是如果能把不同的分区放到不同的磁盘组上，那么在 I/O 负载均衡方面，使用 HASH 分区的作用就要比使用范围分区更为明显了。于是我马上查阅了系统底层存储的设计文档，A 系统的底层存在多个磁盘组：

```
#1  磁盘数：4×36.4   RAID 类型：RAID10   对应 PV：hdisk2   对应 VG：DATA1VG
#2  磁盘数：6×36.4   RAID 类型：RAID10   对应 PV：hdisk3   对应 VG：DATA2VG
#3  磁盘数：6×36.4   RAID 类型：RAID10   对应 PV：hdisk4   对应 VG：DATA3VG
#4  磁盘数：16×36.4  RAID 类型：RAID10   对应 PV：hdisk5   对应 VG：DATA4VG
#5  磁盘数：16×36.4  RAID 类型：RAID10   对应 PV：hdisk8   对应 VG：DATA5VG
```

B 系统使用的是 72 GB 的磁盘，由 20 块磁盘组成 RAID 10，在 VG 划分时进行了条带化处理。在底层存储的设计上，对于 A 系统，如果使用 HASH 分区，可以将 I/O 均匀地分配到 5 个盘组上，从而避免业务高峰期的 I/O 热点。根据以往的经验，这方面的调整可以使性能提升 10%～15%。

在 ID_HINT 列上，使用了一个序列（SEQUENCE）ID_SEQ，这个序列的 CACHE 设置为 1000。一般来说，设置为 1000 就够用了，不过我们还需要进一步测试，看看将 CACHE 设置得更高是否可能带来性能的提升。

从刚才的分析来看，目前可以达到的系统性能提升为 $1-(1-0.1)\times(1-0.05)\times(1-0.1)=23\%$。这是一个比较保守的数字，实际的提升比例可能更高，因此通过这些方面的调整达到预定的优化目标还是很有可能的。

实际上，对于 INSERT 操作来说，使用 BULK INSERT 操作可以大幅度提升插入的性能，不过这样需要对应用进行较大的修改。昨天在讨论方案时，开发人员认为要在短时间内完成修改比较困难，但鉴于客户也希望把 BULK INSERT 作为最后的解决方案，他们会在下一步的优化工作中进行修改，但是不列入本次优化的范围。

今天是在分析问题的过程中度过的，感觉过得很快。快下班的时候，老方打电话过来，说飞机在武汉晚点，刚到北京。我告诉他，经过一天的分析，我认为通过昨天考虑的几方面的优化，使性能提升 30%还是可以实现的。老方听后长舒了一口气，在飞机上他觉得自己在这个项目的售前阶段过于自信，没有做深入的分析，一直担心最终不能让客户满意。

和老方通完电话，我给客户的余经理打了个电话，问她能不能安排一个业务不是很忙的时间，让我做一些针对性的实验，以便于确定优化方案。余经理说这是一个后台系统，我可以随便进行测试，只要不让话单积压过于严重，并确保话单不丢失就可以了。于是我连忙联系维护工程师，和他讨论如何避免系统话单积压。维护人员说他们本身就有一个实时监控系统，如果话单缓冲区占用率超过 60%，就会产生告警，只要缓冲区保持在 60%以下，风险就不是很大。于是我们约定好明天白天就开始进行一系列实验，一旦系统出现风险，他们会马上通知我停止实验。

11 月 26 日　令人惊讶的结果

最近系统的业务量不大，所以客户也同意我们白天进行测试。为了防止误操作，客户把

SCOTT 账号提供给我。今天准备进行以下几个测试：

- SEQUENCE 缓冲区测试；
- 表的 FREELISTS 和 INITRANS 参数调整的测试；
- HASH 分区测试；
- 提交批量测试，测试批量大小对插入性能的影响，分别测试批量为 800、1500、3000、5000 条记录的响应时间；
- BULK INSERT 操作测试。

进行这种单条 SQL 执行时间很短的测试，最好的办法是使用 PROFILER 工具，将要测试的内容写在一个存储过程里，通过 PROFILER 工具来计算平均执行一次所消耗的时间。我首先为每个测试项目都编写了一个小的 PL/SQL 过程，然后开了 7 个终端，运行这个存储过程，在第 8 个终端上的测试过程与其他不同，增加了 PROFILER 的脚本，这样就能够很方便地采集到每条 SQL 的执行情况了。

首先测试序列，我分别对各种 CACHE 值进行了测试，编写了一个测试用的存储过程，如代码清单 10-2 所示。

代码清单 10-2

```
create or replace procedure testSeq(N integer)
is
 i integer;
 b integer;
 v varchar2(20);
begin
  i:=0;
  v:=to_char(sysdate,'yyyy-mm-dd:hh24:mi:ss');
  dbms_output.put_line(v);
  loop
    exit when i>N;
    i:=i+1;
    select sm_idseq.nextval into b from dual;
  end loop;
  v:=to_char(sysdate,'yyyy-mm-dd:hh24:mi:ss');
  dbms_output.put_line(v);
end;
/
```

在第 8 个终端上，执行下面的脚本：

```
declare
    err number;
begin
   err:=DBMS_PROFILER.START_PROFILER ('test seq 1000');
   testseq(200000);
   err:=DBMS_PROFILER.STOP_PROFILER ;
end;
/
```

脚本执行结束后，可以通过代码清单 10-3 所示的脚本查看存储过程中每一行执行的情况。

10

代码清单 10-3

```
column RUN_COMMENT format a40 truncate;
select runid, run_date, RUN_COMMENT from plsql_profiler_runs order by runid;
column unit_name format a15 truncate;
column occured format 999999 ;
column line# format 99999 ;
column tot_time format 999999.999999 ;

select p.unit_name, p.occured, p.tot_time, p.line# line,
       substr(s.text, 1,75) text
  from
       (select u.unit_name, d.TOTAL_OCCUR occured,
               (d.TOTAL_TIME/1000000000) tot_time, d.line#
          from plsql_profiler_units u, plsql_profiler_data d
         where d.RUNID=u.runid and d.UNIT_NUMBER = u.unit_number
           and d.TOTAL_OCCUR >0
           and  u.runid= &RUN_ID) p,
       user_source s
 where p.unit_name = s.name(+) and  p.line# = s.line (+)
 order by p.unit_name, p.line#;
```

其中，参数 run_id 来自于 plsql_profiler_units，这里可以通过我们执行 PROFILER 时使用的名称来查找刚才测试对应的 run_id。一般来说，还有一种更简单的查找方法，就是找最后一个 run_id，因为 run_id 是通过序列产生的，我们刚刚做过的测试肯定是最后一个。

上述查询的结果如下：

```
UNIT_NAME      OCCURED    TOT_TIME  LINE                TEXT
 ----------- ---------- ----------- ----- ---------------------------------------
<anonymous>       1       .000781    4
<anonymous>       1       .009996    5
<anonymous>       1       .002121    6
TESTLOG           1       .001145    6   i:=0;
TESTLOG       50001     40.602085    8   exit when i>=N;
TESTLOG       50000     59.097742    9   i:=i+1;
TESTLOG       50000   2948.819922    10  select sm_idseq.nextval into v from dual;
```

PROFILER 可以计算出每一行执行的次数，以及总共消耗的时间，从而为我们提供准确的测试数据。SEQUENCE 缓冲区的测试结果如表 10-1 所示。

表　10-1

CACHE	并发数量	执行次数	测试时间（毫秒）	平均每次执行时间
2	8	200001	942122	4.7
100	8	200001	73282	0.366
1000	8	200001	48250	0.241
5000	8	200001	47129	0.236
20000	8	200001	43299	0.216

从测试结果来看，SEQUENCE 缓冲区的增加可以提高其访问性能，不过超过 1000 后，再加大 CACHE 参数，性能提升幅度不大。由于目前已经达到了 1000，因此不需要再加大了。

第一项测试结果虽然也在预料之中，但还是让我感到有点失望。哪怕能提升 2%、3%也是好的啊，看样子只能寄希望于后面的测试项目了。第二项测试的是 sm_histable 表的核心参数，测试比对的是一张完全按照目前参数创建的测试表，和修改了 FREELISTS、INITRANS、INITIAL、NEXT 这几个参数的测试表。表的分区方式，以及存储的表空间等属性都没有修改，索引也完全按照生产环境创建。这次测试的结果让人感到十分惊诧，如表 10-2 所示。

表　10-2

项　　目	相关业务	调整前（秒）	调整后（秒）	对比说明
整体时间	短信历史记录应用：DB_DaeMon程序	128.67	112.35	速度提升：14.53%
平均每条记录的插入时间		0.002573	0.002247	速度提升：14.53%

调整这几个参数后，并发插入的性能居然提升了 14.5%，这有点出乎我的预料，基本上达到了我预期的最高值。兴奋之余，我马上进行了 HASH 分区的测试，修改了这张表的定义，将表分区从 10 个修改为 8 个，分别存储在 4 个表空间上，这 4 个表空间分别属于不同的 RAID 组。我并没有将 5 个 RAID 组全部使用，因为在另外一个 RAID 组上，存放了 REDO LOG 文件，这样设计是为了达到 REDO LOG 和数据文件互不干扰的目的。

```
PARTITION BY HASH (ID_HINT)
 PARTITIONS 8
 STORE IN (CQYDSMSC_CENTER1,CQYDSMSC_CENTER2,CQYDSMSC_CENTER3,CQYDSMSC_CENTER4)
```

测试的结果如表 10-3 所示。

表　10-3

项　　目	相关业务	调整前（秒）	调整后（秒）	对比说明
整体时间	短信历史记录应用：DB_DaeMon程序	90.397	78.793	在存储参数提升14.53%的基础上,再提升14.73%
平均每条记录的插入时间		0.00181	0.00158	在存储参数提升14.53%的基础上,再提升14.73%

仅这两项优化，性能总体的提升就已经达到了 27%，这大大出乎了我的意料。按照以上测试结果，仅依靠这两项调整，就可以完成这个项目了。刚才这两项测试都是在 A 系统上进行的，我又在 B 系统上进行了相同的测试，测试结果也令人满意，表核心参数调整后性能提升了 15.5%，不过第二项 HASH 分区的性能提升没有 A 系统高，只有 10.27%，但是总体性能提升也接近了 24%。

上午的实验时间过得很快，不知不觉已经快一点钟了，突然感觉周围静悄悄的，抬眼一看，刚才还坐了 30 多人的办公室里除了我已经没有别人了。这个时候才感觉有点饿，于是放下正在进行的实验，出去吃饭。在路上，我还在回味着刚才的实验，心情十分兴奋。突然看到路边有个

麻辣烫的摊子，想想吃饭的地方离这里还挺远，不如就随便在小摊上吃点，回去继续完成实验。说实在的，我对重庆的麻辣风味还是不太习惯，吃了一小盘就觉得舌头都已经彻底麻木了。

回到办公室，我碰到了现场工程师，问他上午系统是否有异常，他说并没有发现什么异常。听到这样的消息，我的心就更踏实了。上午每次测试的时间都持续了近 20 分钟，如果这对主生产系统产生的影响不大，就说明整个系统的处理能力是足够的。

下午的测试没有取得太多的成果，尽管我将一个插入任务的记录数从 800 条一直提升到了 5000 条，但对性能的提升却微乎其微，原本以为插入任务的记录数在这套系统中是通过参数控制的，可以不修改程序就进行调整，但现实情况似乎并不是这样。根据我以往的经验，一次提交的记录总数有一个最优值，一般在这个最优值下加大记录数，总体插入性能会有所提高，但超过这个最优值，插入性能反而会逐渐下降。这个最优值和数据库的整体情况以及参数配置、REDO LOG 性能等相关。在一般情况下，这个值应该是超过 1000 的。看样子技术人员在系统上线前已经做了充分的测试，选择了一个相对合理的值，因此在这一点上，确实没有多大的优化余地了。

于是，我接着进行了 BULK INSERT 操作的测试。根据以往的经验，BULK INSERT 对于性能的提升是很大的。由于 BULK INSERT 只是作为今后改进的建议，不作为本次优化的重点，因此原本考虑仅通过调整 BULK INSERT 就达到优化目标的想法也只能作罢了。我编写了一个 BULK INSERT 的测试脚本，如代码清单 10-4 所示。

代码清单 10-4

```
CREATE OR REPLACE PROCEDURE TESTFORALL (N INTEGER)
IS
  TYPE T_SM_ID    IS TABLE OF          NUMBER(10)   INDEX BY BINARY_INTEGER;
  TYPE T_SM_SUBID IS TABLE OF          NUMBER(3)    INDEX BY BINARY_INTEGER;
  TYPE T_ORGADDR  IS TABLE OF          VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_DESTADDR IS TABLE OF          VARCHAR2(21) INDEX BY BINARY_INTEGER;
  TYPE T_ID_HINT  IS TABLE OF          NUMBER(10)   INDEX BY BINARY_INTEGER;
  V_SM_ID         T_SM_ID;
  V_SM_SUBID      T_SM_SUBID;
  V_ORGADDR       T_ORGADDR;
  V_DESTADDR      T_DESTADDR;
  V_ID_HINT       T_ID_HINT;
  I INTEGER;
BEGIN
   FOR I IN 1.. N
   LOOP
      V_SM_ID(I):=I;
      V_SM_SUBID(I):=12;
      V_ORGADDR(I):='444555565';
      V_DESTADDR(I):='555555';
      SELECT SM_IDSEQ.NEXTVAL INTO V_ID_HINT(I) FROM DUAL;
   END LOOP;
   FOR I IN 1..N LOOP
     INSERT INTO SM_HISTABLE0101 (SM_ID,SM_SUBID,ORGADDR,DESTADDR,ID_HINT) VALUES
       (V_SM_ID(I),V_SM_SUBID(I),V_ORGADDR(I),V_DESTADDR(I),V_ID_HINT(I));
   END LOOP;
   COMMIT;
   FORALL I IN 1..N
```

```
      INSERT INTO SM_HISTABLE0101 (SM_ID,SM_SUBID,ORGADDR,DESTADDR,ID_HINT) VALUES
        (V_SM_ID(I),V_SM_SUBID(I),V_ORGADDR(I),V_DESTADDR(I),V_ID_HINT(I));
    COMMIT;
END;
/
```

测试结果不出所料，性能提升了数倍，如表 10-4 所示。

表　10-4

项　　目	相关业务	调整前（秒）	调整后（秒）	对比说明
整体时间	短信历史记录应用：DB_DaeMon程序（A系统）	0.149	0.0389	速度提升3.83倍
平均每条记录的插入时间	0.000186	0.000048	速度提升3.83倍	
整体时间	短信历史记录应用：DB_DaeMon程序（B系统）	0.0911	0.0247	速度提升3.68倍
平均每条记录的插入时间	0.0001138	0.000031	速度提升3.68倍	

做完实验，刚刚下午三点多。我联系了余经理，把实验的结果告诉了她。她听后十分高兴，建议马上开会，如果测试的结果得到确认，明天就可以调整表结构，测试性能。

我马上用刚才的测试数据做了一个简单的 PPT，赶到会议室已经快四点了，余经理和工程师正在会议室里等着我。看到我演示的结果，他们都十分兴奋。由于这套系统使用了日表，因此只要重建明天使用的表就可以完成优化工作了。为了确保安全，后天和以后的表暂时不做调整，等明天使用结果出来后再决定是否统一修改日表创建的脚本。

11 月 27 日　更大的意外

昨天晚上调整了表结构，我建议对 REDO LOG 也相应做些调整。目前 A 系统的 LOG BUFFER 是 1 MB，REDO LOG 文件大小是 61 MB；B 系统的 LOG BUFFER 是 120 KB，REDO LOG 文件大小是 100 MB，并且 B 系统的每个日志组有两个成员。对于这种 REDO 量十分高的系统，使用两个成员虽然增加了可靠性，但是对性能的负面影响也是很大的。因此我建议将 B 系统的 REDO LOG 成员改为 1 个。

昨晚调整建表脚本的同时，我还针对数据库进行了一些调整，其中对 SGA 和 REDO 的调整如表 10-5 所示。

表　10-5

参　　数	当　前　值	建　议　值	设置原则
DB CACHE SIZE	NP:3072M RH:2048M	NP:4096M RH:2048M	对于本系统，在内存充足的情况下，设置DB Cache，尽量使DB BUFFER的命中率提高。如果B系统扩容后内存充足，增加DB Cache的大小，可以提高查询和单条INSERT的性能（如不使用FORALL操作）
DB KEEP SIZE	未设置	NP:300M RH:300M	将HISTABLE以外的常用对象放入KEEP池

（续）

参　数	当 前 值	建 议 值	设置原则
LOG BUFFER	NP: 1024000 RH: 124416	NP:20M RH:20M	设置LOG BUFFER的原则是使系统在忙时不出现log buffer space等待
SHARED POOL SIZE	NP: 314572800 RH: 318767104	NP:400M RH:400M	设置比较合理的时候，不会出现较多的库缓存和SHARED POOL闩锁等待
REDO LOG文件	NP:61M RH:100M	2000M	业务高峰期，减少日志切换

A 系统的 REDO LOG 文件以前存放在 HDISK2 上，而 UNDO、TEMP 以及一部分索引所在的表空间也使用了 HDISK2，为了减少 I/O 方面的争用，应将这些索引存放在其他表空间上，同时把 UNDO 表空间移到 HDISK3 上。

对于 SM_HISTABLE 参数做了如下调整。

A 系统和 B 系统的 SM_HISTABLE 表的存储参数设置建议如表 10-6 所示。

表　10-6

参　数	当 前 值	建 议 值	说　明
FREELISTS	1	8	不小于最大并发插入进程的数量
INITIAL	64 KB	8 MB	
NEXT	64 KB	8 MB	
INITRANS	未设置	8	

A 系统和 B 系统的 SM_HISTABLE 表索引的存储参数设置建议如表 10-7 所示。

表　10-7

参　数	当 前 值	建 议 值	说　明
FREELISTS	1	8	不小于最大并发插入进程的数量
INITIAL	64 KB	4 MB	最小设置为2 MB
NEXT	64 KB	4 MB	最小设置为2 MB
INITRANS	未设置	8	

昨晚停了 1 个小时数据库，还好开发人员设计的短信平台很灵活，数据库停止后，所有的话单被暂存在内存中，内存缓冲区存满后会转储到文件中，因此停数据库对短信业务并无影响。数据库重启后，这些短信话单被集中写入，事后经过业务部门的确认，最高峰时，A 系统和 B 系统短信话单的插入数量达到了惊人的 5800 条/秒，而从系统监控上看，CPU 负载不到 50%，I/O 负载也只有 40%左右，系统还是有很大潜力的。以这样的性能情况来看，这个项目应该可以圆满结束了。

按照以往的惯例，做完实施后的第二天我一大早就到了客户现场，系统运行得很平稳，CPU 使用率在 20%以下，I/O 负载也很小。这是我意料之中的事情，于是我打电话和余经理通报了一下今天的情况，然后进行了两项测试，第一项是创建一张和原来一样的表，测试调整了 REDO 后，性能有多大的提升，测试结果让我感到十分惊讶，如表 10-8 所示。

表　10-8

基线1（RUNID 10）	模拟目前A系统的REDO LOG设置	
指标或参数	**数值**	**备注**
LOG BUFFER	1000 KB	
日志组数量	8	
日志文件大小	60000 KB	
表分区方式	范围	
执行次数	50000	
执行时间	37968	
平均每个INSERT的时间	0.75936	
日志切换时间	11秒	
主要等待事件	log buffer space	
	buffer busy wait	
	log sync	
基线2 (RUNID 11)	根据Oracle建议优化后的测试基线	
指标或参数	**数值**	**备注**
LOG BUFFER	20 MB	
日志组数量	3	
日志文件大小	1500 MB	
表分区方式	范围	
执行次数	50000	
执行时间	24189	
平均每个INSERT的时间	0.48378	提高57%
日志切换时间	5分钟	
主要等待事件	buffer busy wait	

本次测试模拟了实际生产环境中写入进程的工作，先启动 7 个插入进程进行插入操作，在第 8 个进程中使用 PROFILER 进行跟踪，每个批次插入 800 条记录，循环 50000 次。优化前和优化后的性能对比十分惊人，居然达到了 57%，这是我没有预想到的。后来我又在 B 系统上做了同样的测试，虽然性能提升没有 A 系统那么明显，不过也达到了 38%。

我马上给北京的老方打了个电话，告诉他这里的优化效果。老方感到十分惊讶，这个项目真可谓是柳暗花明，原本以为很难搞定的项目居然通过简单地调整了一些参数就取得了这么大的效果。

下午的总结会上，大家的心情都很不错。小余望着窗外久违的阳光说："按理说，这么好的天气就不该上班了，应该找个茶馆喝茶去。"大家听完都乐了。我说："我们干脆找个茶馆开会算了。"任务完成得不错，大家的心情当然就好了。

这个优化项目很成功，第二年除夕的时候，我收到了小余的一条短信："今天系统一切正常，谢谢你，祝你新年愉快。"春节时收到千里之外的祝福，确实是件很惬意的事情。

第 11 章

理解索引

11

索引是什么？恐怕很多 DBA 都没有彻底理解索引的真正含义。为什么有时候通过索引访问一张表会比较快呢？想要回答这些问题，需要从头了解索引到底是什么。大家小时候应该都用过新华字典，它是按照拼音字母排序的，因此查字典时可以通过汉字的拼音来查找。最简单的方式是根据拼音字母的大体位置先随便翻开一页，然后根据这一页的内容再次翻页，直到找到这个汉字。这种翻字典的方式有点土，速度也比较慢，不过这是我们使用的一种最原始的索引方式。为了提高查字典的速度，我们一般都会在字典侧面标出某个字母所在的位置，这样就可以首先根据字母所在的位置，更为精确地判断某个字可能的位置。这种包含两级的索引，加快了定位的速度。

实际上，Oracle 的索引访问和刚才看到的翻字典是类似的。Oracle 的索引是一种 B 树结构，学过《数据结构》这门课的朋友可能对 B 树比较熟悉，既然是树，就应该有树根、树枝和树叶。对一棵树的访问肯定要从根出发，然后经过树枝，最终到达树叶，如图 11-1 所示。

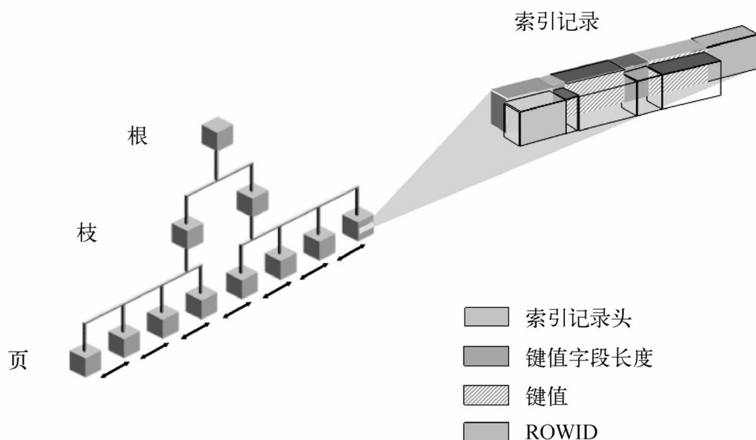


图　11-1

如果我们要查找一条 ID=3 的记录，而在 ID 字段上有索引，那么就可以通过索引，直接找到 ID=3 的所有索引项，然后根据这些索引项逐条从表中找出所有需要的字段，这就是常见的索引访问方式。

从索引的原理可以看出，通过访问索引可以提高系统的访问速度。但是通过索引访问就一定会提高性能吗？答案当然是否定的。大家都知道数据库访问数据的主要开销分为 I/O 开销和 CPU 开销两部分。通过一次需要访问的数据块的总数量，就可以初步判断出操作的大体开销。如果有一个查询，需要查出一张有 3000 万记录的表中的 2000 万条记录，那么通过索引访问这张表可能就会比直接对这张表做全表扫描慢许多。鉴于索引访问的这个特点，我们不能想当然地认为索引访问就是好的。

从上面的讨论我们已经知道了，索引其实并不神秘，它只是存储在数据库里的树状结构的数据，我们可以通过这些数据提高某些表访问的性能。那么下一个问题就是怎么在应用中设计合理的索引，从而达到最好的效果呢？可能有朋友要说了，既然索引有这么好的效果，那么给每个字段都创建一个索引不就可以了。刚才我们说了，索引是数据库中的一种特殊存储结构，当表中的数据变化时，索引是必须做同步更新的，因此索引带来的并不仅仅是查询性能的提升，还会带来一个副作用：索引的更新是需要成本的，过多的索引可能会带来对写入操作性能的负面影响。基于这个原理，我们在设计索引时需要统筹考虑，尽可能用最少的索引达到最佳的效果。事实上，应用系统在设计索引时，只有很少的索引是经统筹考虑后创建的，大多数索引都是系统运行过程中随意添加的。一个系统长时间运行后，系统中的索引就十分混乱了。我在做性能优化时经常会碰到一些表存在大量的索引，五六个算少的，多时能达到十多个。这些索引都是在出现了一些性能问题后，为了单一解决某个问题而添加的。一张表中存在这么多的索引，仅维护索引所增加的那些开销已经不能解决问题了。最为关键的是这些表上还存在很多与索引字段相近的索引，系统经常会由于分析数据不准确而出现索引选择错误的现象，从而导致系统性能极为不稳定。

在这里，我又想到了一个十分著名的问题，一张表上到底设计多少个索引比较好？经常有朋友问我这个问题，而且也有很多文章和书籍给出了十分明确的数字——一张表上最好不超过 6 个索引。但我不知道这个数字是怎么得出的，因为我从来没有在任何官方资料或者学术性论文中看到过这样的描述。好像 TOAD 里有一个简单的数据库健康检查工具，其中有一项是检查索引超过 6 个的表，也许某些 DBA 就认为超过 6 个索引的设计可能是有问题的，而少于 6 个一般不会有问题。实际上仅用这个数字来判断索引设计是否合理是十分不恰当的，我曾经见到过存在十多个索引的表，但是这些索引都是必需的；也有些表只有两三个索引，但是索引的设计却存在问题。我想 TOAD 中的这个工具只是为了提醒大家系统中一些表上的索引过多，需要检查索引设计是否合理，而并不是将 6 个索引作为判断合理性的分界线。另外要说明的一点是，虽然索引会增加维护的成本，影响 DML 语句的性能，但是在一般的 OLTP 系统中，和 DML 操作相比，SELECT 操作所占的比例要高得多，大多数系统中这一比例都高达 80%，有的甚至能达到 90%，在这种系统中，如果少一个索引，可能导致某张大表经常进行全表扫描，增加的 CPU 和 I/O 开销可能达到这个索引维护成本的几十倍甚至上百倍。如果你明白了这一点，就会知道判断索引是否合理不仅仅是一个数字这么简单的。

一般分析索引是否合理的方法是将和某张表相关的 SQL 都查找出来，按照 BUFFER GET 或者 PHYSICAL READ 排序，分析排在前面的对系统性能影响较大的 SQL，从中找出 WHERE 条件和连接条件，从而判断如何创建索引才更为合理。这是一项十分艰苦的工作，不仅仅需要技术，

更需要认真的态度和坚韧的精神。使用这种方法得出的索引设计原则是比较合理的。随后我将通过案例来详细介绍分析索引的方法，这里暂不对该方法进行深入的讨论。

11.1　反转键索引的误区

在回顾了索引的一些基础知识后，我们需要进一步了解索引优化的相关知识，研究索引的主要种类以及各类索引在使用时的一些要点。

首先来看最常见的 B 树索引。B 树索引适用于几乎所有的场合，也是系统中使用最为广泛的索引形式。实际上，我们所说的普通索引、反转键索引、降序索引、函数索引等都是 B 树结构的，其物理存储结构是完全相同的。与之相对的位图索引则是完全不同的存储结构，位图索引不是树状结构，没有枝节点，只有叶节点。B 树索引操作包括索引唯一性扫描、索引范围扫描、快速索引全扫描和索引全扫描，而位图索引的访问方式只有一种，就是索引全扫描。在使用位图索引时，只有对索引完全扫描一遍，才能找到所有需要的行。

我们知道，索引是一种树状结构，其组织形式是一棵扩展了的 B 树，和普通 B 树不同的是，这棵 B 树的所有叶节点上都有一条双向链，称为叶节点链。这条双向链是根据索引键值的大小进行排序的，它的存在十分重要，是实现索引范围扫描最关键的技术。当进行索引范围扫描时，首先通过 B 树的定位算法，从根开始，找到范围扫描起始键值的位置，然后从这个位置开始，通过叶节点链按照升序或者降序的方式扫描相关的叶节点，直到找到超出扫描范围的键值为止。

最为普通的索引是按照键值升序排列的，索引树右面枝叶的键值总比左边的大。而如果我们设计了降序索引，那么情况则正好相反。

函数索引是一种特殊的 B 树索引，引入函数索引的目的是为了解决那些在使用过程中必须在字段上做函数运算的情况。一般情况下，我们会建议开发人员不要在 WHERE 条件中的表字段上使用函数，以便为其设计索引。不过事实上，我们无法杜绝这样的函数存在，比如在某些情况下，必须从一个字段取第 2、3 位进行比较：WHERE substr(id,2,2)='ID'，如果这样的查询条件放弃函数的话，程序员的处理将十分复杂。函数索引为这种情况提供了很好的帮助，如果这个查询条件使用索引效果较好的话，我们就可以在 ID 字段上创建一个以函数 substr(id,2,2) 为键的 B 树索引。事实上，在绝大多数应用系统中，函数索引都是不可避免的，但不幸的是，在我做过的优化项目中，我基本上没有看到过用户使用这种索引。

接下来讨论反转键索引（ reverse key index ），这是一种十分著名的索引。反转键索引在存储键值时，先将键值进行翻转，比如，'1234'存储在索引中的键值是'4321'。设计反转键索引的目的是为了解决索引的热块冲突问题。索引块出现热块冲突是我们在性能优化时经常会碰到的问题，比如一个主键是通过序列生成的，那么主键索引就可能成为热块，这种情况下，如果我们确定针对主键的查询不存在或者很少有索引范围扫描，那么就可以考虑使用反转键索引来解决主键的热块冲突问题。其原理很简单，通过键值的反转，打乱索引数据块中的数据组织，从而将热点数据分散到不同的索引数据块中。

除了解决热块冲突的问题外，DBA 界还流传着这样的说法：反转键索引可以解决 like '%abc'

无法使用索引的问题。粗想起来，确实是这样，由于通配符在第一个字符，like '%abc'这样的条件无法进行索引范围扫描，因此一般情况下使用全表扫描比较合适。不过在某些特殊情况下，如果表规模巨大，这种全表扫描的成本会很高，如果能使用索引就好了。而反转键索引正好在存储键值时是反转过来的，1abc,2abc 在索引键中的存储为 abc1,abc2，这种情况下，可以通过范围扫描将符合条件的记录找出来吗？这种解释似乎是很合理的，我也曾经被这种理论蒙蔽过一段时间，直到有一天我自己做了一个实验，才发现问题远非那么简单。下面就来回顾一下这个实验，首先创建测试表：

```
DROP TABLE TINDEX;
CREATE TABLE TINDEX as SELECT DISTINCT
OBJECT_NAME,OBJECT_ID,OBJECT_TYPE,CREATED,STATUS,TEMPORARY,TIMESTAMP,'ABCDEFGHIJKL
MNOPQRSTUVWXYZ1234456' abc
from dba_objects;
INSERT INTO TINDEX SELECT DISTINCT
OBJECT_NAME,OBJECT_ID,OBJECT_TYPE,CREATED,STATUS,TEMPORARY,TIMESTAMP,'ABCDEFGHIJKL
MNOPQRSTUVWXYZ1234456' abc
from dba_objects;
create index  idx_tindex_name on tindex(object_name ) reverse;
```

然后做一次表分析：

```
exec DBMS_STATS.GATHER_TABLE_STATS(ownname=>'scott', tabname=>'tindex',
estimate_percent=>30, -cascade=>true, degree=>2);
```

似乎一切都准备好了，下面来测试反转键索引是否真的能够解决 like 语句的问题：

```
SQL> set autotrace traceonly
select *  froM tindex where object_name like '%TINDEX';
SQL>

Execution Plan
----------------------------------------------------------
Plan hash value: 2264840918

----------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |  5297 |   553K|   103   (3)| 00:00:01 |
|*  1 | TABLE ACCESS FULL | TINDEX |  5297 |   553K|   103   (3)| 00:00:01 |
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("OBJECT_NAME" LIKE '%TINDEX')

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
       1717  consistent gets
```

```
          0  physical reads
          0  redo size
        939  bytes sent via SQL*Net to client
        400  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

似乎索引并没有被自动使用，我们使用 hint 强制索引来看看：

```
SQL> select /*+ INDEX(TINDEX idx_tindex_name ) */ *  from tindex where object_name like
'%TINDEX';

Execution Plan
----------------------------------------------------------
Plan hash value: 2021627753

---------------------------------------------------------------------------------
|Id |Operation                  | Name          | Rows | Bytes |Cost (%CPU)|  Time  |
---------------------------------------------------------------------------------
| 0 |SELECT STATEMENT           |               | 5297 | 553K  | 5548  (1) |00:00:54|
| 1 |TABLE ACCESS BY INDEX ROWID|TINDEX         | 5297 | 553K  | 5548  (1) |00:00:54|
|*2 |INDEX FULL SCAN            |IDX_TINDEX_NAME| 5287 |       |  520  (1) |00:00:05|
---------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("OBJECT_NAME" LIKE '%TINDEX')

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        520  consistent gets
          0  physical reads
          0  redo size
        939  bytes sent via SQL*Net to client
        400  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

这里虽然使用了索引，但扫描方式是全索引扫描，而不是我们期待的索引范围扫描。看来反转键索引并不能解决这个问题。我们继续下面的实验：

```
SQL> create index idx_tindex_func on tindex(reverse(object_name));

Index created.
```

```
SQL> exec DBMS_STATS.GATHER_TABLE_STATS(ownname=>'scott', tabname=>'tindex',
    estimate_percent=>30, -> cascade=>true, degree=>2);

PL/SQL procedure successfully completed.
SQL> select *  from tindex where reverse(object_name) like 'XEDNIT%';

Execution Plan
----------------------------------------------------------
Plan hash value: 1286384425

---------------------------------------------------------------------------------
| Id |Operation                   |Name           | Rows |Bytes|Cost (%CPU)|  Time  |
---------------------------------------------------------------------------------
|  0 |SELECT STATEMENT            |               |  4   | 436 | 7     (0) |00:00:01|
|  1 |TABLE ACCESS BY INDEX ROWID|TINDEX         |  4   | 436 | 7     (0) |00:00:01|
|* 2 |INDEX RANGE SCAN            |IDX_TINDEX_FUNC|  4   |     | 3     (0) |00:00:01|
---------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(REVERSE("OBJECT_NAME") LIKE 'XEDNIT%')
       filter(REVERSE("OBJECT_NAME") LIKE 'XEDNIT%')

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
        939  bytes sent via SQL*Net to client
        400  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

这才是我们需要的效果，通过 reverse 函数，然后将%TINDEX 反转为 XEDNIT%，这样才能真正解决问题，而这个 SQL 的开销是原 SQL 的几百分之一。实际上，这种解决方案只能在修改应用的前提下实现，不如使用反转键索引那么简单、用途广泛，不过我们也终于通过实验纠正了流传甚广的错误。我通过搜索引擎查阅了大量关于此话题的英文资料，终于明白了这一误解的来源，最初，确实有一篇文章介绍如何使用 reverse 函数来解决这个问题：

```
SELECT *
FROM customer
WHERE reverse(Cust_Name) LIKE '%saliV%';
```

但后来这篇文章在被转载时，错误地写成了：

```
SELECT *
FROM customer
WHERE Cust_Name LIKE '%saliV%';
```

以此为蓝本，终于引发了通过反转键索引优化 like 操作这个错误的观点。由此可见，不经过自己验证就全盘吸收网络上的知识是多么危险。

不过 Oracle 反转键索引的存储结构，确实具备对 Like 条件做范围扫描的基础，这种扫描和 Oracle 以往提供的任何一种索引扫描技术都不相同，是一种全新的索引扫描方式。也许在 Oracle 12 或者 13 版本中，真的会出现类似的功能，Oracle 的每个新版本给大家带来的惊喜一直都是出乎大多数人意料的。

11.2　索引访问的方式

设计索引是专业性很强的工作，一般来说，索引设计最好有专业 DBA 参与，不过现实中，绝大多数系统的索引设计都是由开发人员完成的。由于开发人员对索引的原理不甚了解，因此大多数软件系统中的索引设计方面都存在很大的问题。Oracle 公司有一门培训课程叫做“开发 DBA”，这门课程的主要目的是让开发人员理解一些 Oracle 的基本原理与概念，从而在设计应用系统时依据这些原理，避免出现严重的问题。在开发人员中进行类似“开发 DBA”这样的课程培训，是数据库优化工作中十分重要的环节，而这个环节往往被我们所忽视。实际上，对研发团队进行适当的培训，可以节省大量的成本。

本节要讨论的话题是如何设计索引，这是很多 DBA 都想了解的。实际上，前面章节已经讨论过设计索引的一些基本方法。索引的设计要遵循一些最基本的原则。首先必须有针对性，索引的目的是快速定位数据，因此每个索引都必须符合快速定位数据的要求。比如，我们想让下面的这条语句执行得更快一些，消耗更少的资源。

```
select emp_name,emp_id,sal from emp where emp_id=11023
```

首先需要分析 emp 这张表，emp 表是企业的职员表，有数万条记录，而每个职员都有唯一的 emp_id 对应，如果在 emp_id 上创建了索引，那么这条 SQL 就可以通过索引快速定位到 emp_id，然后通过索引中的 ROWID 直接找到 emp 表中的相关记录，并取出所需要的数据。这种情况下，在 emp_id 上创建一个索引就是十分必要的。我们再来看下一个语句：

```
select emp_id,sal from emp where emp_id=11023 and emp_name like 'John%'
```

在这个语句中，WHERE 条件有两个：一个是 emp_id=11023，另外一个是 emp_name like 'John%'。这时 Oracle 该如何来选择合适的执行计划呢？这里讨论的前提是使用 CBO 优化器（RBO 优化器在索引选择时智能程度很低，出现执行计划错误的机会要比 CBO 大得多，因此建议在条件允许的情况下，尽可能使用 CBO），CBO 会根据索引的统计数据计算每个索引访问路径的成本，从而选择一种开销较小的执行计划。在这个案例中，如果通过 emp_id 来查找相关的数据，emp_id 就是主键，根据 emp_id=11023 可以唯一定位到一条记录，然后就可以取出这条记录中 sal、emp_name 和 emp_id 这三个字段的值，将 emp_name 字段的值和 WHERE 条件中的 emp_name like 'John%'进行比较，如果这条记录符合此条件就返回该记录，如果不符合则没有记录返回。

下面来分析另外一条访问路径。如果使用 emp_name 上的索引，首先应找出所有的 emp_name like 'John%'记录，假设能找到 10 条，接着通过索引中提供的这 10 条记录的 ROWID 去访问这张表，把 emp_id、sal 这两个字段的值取出来，然后再通过另外一个过滤条件 emp_id=11023 进行筛选，把符合条件的记录找出来。很明显，这条访问路径的开销要比第一条略高，因为这里先查出了 10 条记录，然后又通过过滤器过滤掉了其中的 9 条，而第一种执行计划只检索了一条记录就获得了结果。

第三种访问路径就是我们所说的全表扫描。如果我们在 emp_name 和 emp_id 上都没有创建索引，那么就必须对这张有几万条记录的表从头到尾扫描一遍，找到所有符合上面两个过滤条件的记录，并把对应的 emp_id 和 sal 值检索出来。

这三条访问路径到底哪个开销更小呢？实际上，Oracle 的 CBO 优化器会自动计算访问路径的成本，并且选择最佳路径来执行这条 SQL。上例中包含了 Oracle 访问数据的三种常见的方法。第一种是索引唯一性检索，通过唯一性索引定位到某条记录，读出该记录后通过 emp_name 这个过滤条件进行筛选，最后获得符合条件的所有记录。第二条路径是通过 emp_name 上的索引找出所有符合条件的记录，这种查找方式是找到符合条件的第一条记录，然后顺着叶节点链按照升序或者降序的方式扫描出所有符合条件的记录的 ROWID，再依次将表中的相关数据块读出来，根据其他的过滤条件进行筛选，最后找到符合条件的记录，这种方式就是我们常说的索引范围扫描。第三种方法是直接读取表的数据，将数据读出后，根据 emp_id 和 emp_name 两个条件进行过滤，获得所需要的数据。

另外，还有一种数据访问路径没有在上述示例中出现，这种方法是从 Oracle 9*i* 开始引入的。比如，我们有一个复合索引 IDX_ID_NAME，这个索引包含两个字段 emp_id 和 emp_name，而有这样一条 SQL：

```
select emp_id,sal from emp where  emp_name= 'John'
```

这种情况下，我们可以通过 IDX_ID_NAME 来查找所需要的数据。由于 emp_name 不是这个索引的第一个字段，因此无法像普通的索引唯一扫描（index unique scan）或索引范围扫描（index range scan）那样通过一次定位，然后顺着叶节点链进行扫描。这里的扫描必须是跳跃式的，因为这个索引的前导字段是 emp_id，所以扫描时，对于每个 emp_id，都需要做一次定位，然后通过叶节点链查找到所有符合条件的记录。Oracle 给这种扫描方式起了一个很形象的名字叫做 index skip scan，我们称之为索引跳跃式扫描。正因为如此，分段的数量对于扫描的成本影响很大。本例中 emp_id 是主键，在这种情况下，使用索引跳跃式扫描的成本实际上是比较高的，其开销甚至高于索引全扫描。

我们刚才又引入了另外一种索引访问的方式，即索引全扫描，它包括两种不同的方式：一种称为索引全扫描（index full scan），另外一种称为索引快速全扫描（index fast full scan）。为什么要使用索引全扫描呢？我们先来看一条 SQL：

```
select emp_name from emp where emp_name is not null
```

在 emp_name 上有一个索引，其中包含了所有 emp_name 不为空的值，因此在这种情况下，

只要对整个索引进行一次扫描就可以完成这条 SQL 了，不需要进行全表扫描。一般来说，对于字段较多的表，索引的大小会远小于表的大小，因此全索引扫描的成本会远小于全表扫描。本例可以使用索引快速扫描，根据该索引的扩展，以多块读的方式来进行。因此在这类操作中，我们会看到会话出现大量的 db file scattered read 等待。但如果经常出现 db file scattered read 等待，是不是就说明系统中出现了全表扫描呢？其实这种说法并不准确，因为索引快速全扫描也会以多块读的方式来进行。

如果我们对刚才那条 SQL 进行一些小的修改：

```
select emp_name from emp where emp_name is not null order by emp_name
```

这时，SQL 的执行路径可能会有所改变，将不再使用索引快速全扫描。这是因为索引快速全扫描是根据扩展的顺序进行的，并不是按照 emp_name 值的大小进行的，因此这样扫描出来的 emp_name 数据不会按照 emp_name 排序，如果要排序，必须在索引扫描完成后再进行。这样的执行计划可能不如索引全扫描便捷，这两者之间是存在区别的。索引全扫描是根据叶节点链来进行的，扫描首先从根开始，接下来找到叶节点链上的第一个数据块，然后沿着叶节点链进行。由于叶节点链是根据索引键值排序的，因此扫描出来的数据也是经过排序的，数据读出后不需要再次排序。使用这种扫描方式时，首先要找到索引的根，然后通过枝节点找到第一个叶节点，最后再顺着叶节点链扫描整个索引。另外，索引全扫描的 I/O 成本要比索引快速全扫描大很多，尽管读取根节点和叶节点的成本并不大，但由于顺着叶节点链扫描整个索引时无法进行多块读操作，而只能进行单块读操作，因此会产生非常大的 I/O 开销。在这种索引扫描进行时，如果我们对会话进行跟踪，会发现大量的 db file sequential read 等待。

在这里，大家可能会有一个疑问，带 order by 的 SQL 语句使用索引全扫描的成本是否一定小于索引快速全扫描呢？实际上，这不能一概而论，关键要看排序操作的成本是否大于索引单块扫描与索引多块扫描之间的 I/O 成本差。

谈到这里，大家应该已经基本了解了索引访问的主要方式。实际上，还有一种索引访问的操作本节并没有讨论，就是位图索引的访问。由于位图索引的结构十分特殊，没有枝节点，而是一种平面结构，因此，如果通过位图索引进行扫描，就只有一种扫描方式，这种方式类似于普通索引的索引快速全扫描，根据该索引的扩展，通过多块读进行扫描，这是因为位图索引的每个数据块中都可能包含我们所需要的键值。

11.2.1　小表用索引有意义吗

传统观念认为，小表使用索引意义不大，全表扫描可能具有更高的效率。实际上，我们理解了表访问的方式，就可以来分析这个问题了。如果要访问一张没有索引的表，我们需要读出这张表的所有数据并进行比对，这样才能得到查询的结果。由于每次读取表数据行的数量是有限的，因此访问一张只有三四个数据块的小表可能就需要 2 ~ 30 个 BUFFER GET。

假如我们从一张有 350 条记录的表中查找一条记录，如果这张表的数据都已经在 DB Cache 中了，那么要完成全表扫描就需要产生 8 个 BUFFER GET：

```
--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    79 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T3   |     1 |    79 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
Statistics
----------------------------------------------------------
         50  recursive calls
          0  db block gets
          8  consistent gets
          0  physical reads
          0  redo size
        484  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

而如果我们使用索引访问这张小表，通过根节点马上就可以找到相关的叶节点，然后定位符合条件的记录，最后通过 ROWID 就可以快速定位到要查找的那条记录，这样的查询开销可能会小于全表扫描。

```
----------------------------------------------------------------------------------
| Id  | Operation                   | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |   1   |  79   |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID|  T3  |   1   |  79   |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          |IDX_T3|   1   |       |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        484  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

通过上面的测试，我们又推翻了一个大家以前都承认的“真理”。不使用索引直接访问小表，效率未必是最高的，这和表的记录大小、记录分布有很大的关系。

11.2.2　位图索引为什么不适合大并发量环境

对于某些情况来说，普通的 B 树索引是无能为力的，比如全国所有人的身份证信息表，共有约 13 亿条记录，其中性别这个字段，只有男、女两个取值，对于该字段创建 B 树索引，访问效率是很低的。对于这类情况，Oracle 提供了一种特殊的索引——位图索引。位图索引和普通的 B

树索引有所不同，它没有树状结构，而只有一个类似 HEAP 表的平面结构。位图索引虽然也有枝节点，但它和 B 树索引的枝节点是不同的，没有排序和导航功能，仅仅指出了包含某个键值的数据块的范围，枝节点可以在位图索引值扫描时缩小索引扫描的范围。位图索引的访问方式包括位图索引全扫描（bitmap index full scan）、位图索引值（bitmap index single value）、位图索引快速全扫描（bitmap index fast full scan）等。下面通过一个实验来介绍位图索引的结构。

首先创建一张表 ta，插入一些数据，然后提交，再创建一个位图索引：

```
create table ta (a char(1));
    insert into ta values ('1');
    insert into ta values ('2');
    insert into ta values ('3');
    insert into ta values ('4');
    insert into ta values ('4');
    insert into ta values ('4');
    insert into ta values ('2');
    insert into ta values ('1');
    insert into ta values ('3');
 commit;
 create bitmap index bi_ta on ta(a);
```

接下来，需要使用 ALTER SYSTEM DUMP DATAFILE...BLOCK...语句把位图索引的叶节点内容转储出来：

```
Leaf block dump
===============
header address 181916764=0xad7d45c
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 4
kdxcosdc 0
kdxconro 4
kdxcofbo 44=0x2c
kdxcofeo 7947=0x1f0b
kdxcoavs 7903
kdxlespl 0
kdxlende 0
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8032
row#0[8011] flag: -----, lock: 0
col 0; len 1; (1):  31
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
col 3; len 2; (2):  c8 81
row#1[7990] flag: -----, lock: 0
col 0; len 1; (1):  32
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
```

```
col 3; len 2; (2):  c8 42
row#2[7968] flag: -----, lock: 0
col 0; len 1; (1):  33
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 0f
col 3; len 3; (3):  c9 04 01
row#3[7947] flag: -----, lock: 0
col 0; len 1; (1):  34
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
col 3; len 2; (2):  c8 38
----- end of leaf block dump -----
```

从上面的数据我们可以看出，这里记录了4条位图信息。下面通过第1条记录来分析位图索引的构成。

```
row#0[8011] flag: -----, lock: 0
col 0; len 1; (1):  31
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
col 3; len 2; (2):  c8 81
```

首先可以看出，位图索引没有枝节点，直接是叶节点。这也是位图索引占用空间比较小的主要原因。由于索引较小，所以全索引扫描的效率应该高于B树索引。

再来看索引的结构，我们会发现每组位图是由4个字段组成的。第1个字段表示键值（‘1’）；第2个字段是位图所表示的数据块的RDBA低值；第3个字段是位图所表示的数据块的RDBA高值；第4个字段是位图信息，C8表示位图是8位长的，0X81可展开为10000001，由于位图的顺序是从右向左，因此第1条记录和第8条记录的键值都是‘1’。同理，第2行（键值是‘2’）的位图信息是C8 42，其中0X42可展开为01000010，即第2行和第7条记录的键值是‘2’。

接下来，我们再插入几条记录，来观察这个位图索引的变化。

```
insert into ta values ('3');
insert into ta values ('3');
insert into ta values ('2');
commit;
```

以下是插入了几行数据后位图索引的新内容：

```
Leaf block dump
===============
header address 181916764=0xad7d45c
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 4
kdxcosdc 0
kdxconro 7
kdxcofbo 50=0x32
kdxcofeo 7883=0x1ecb
kdxcoavs 7833
```

```
kdxlespl 0
kdxlende 2
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8032
row#0[8011] flag: -----, lock: 0
col 0; len 1; (1):  31
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
col 3; len 2; (2):  c8 81
row#1[7990] flag: -----, lock: 2
col 0; len 1; (1):  32
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
col 3; len 2; (2):  c8 42
row#2[7883] flag: -----, lock: 2
col 0; len 1; (1):  32
col 1; len 6; (6):  03 02 ad aa 00 08
col 2; len 6; (6):  03 02 ad aa 00 0f
col 3; len 1; (1):  03
row#3[7968] flag: ---D-, lock: 2
col 0; len 1; (1):  33
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 0f
col 3; len 3; (3):  c9 04 01
row#4[7925] flag: ---D-, lock: 2
col 0; len 1; (1):  33
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 0f
col 3; len 3; (3):  c9 04 03
row#5[7903] flag: -----, lock: 2
col 0; len 1; (1):  33
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 0f
col 3; len 3; (3):  c9 04 07
row#6[7947] flag: -----, lock: 0
col 0; len 1; (1):  34
col 1; len 6; (6):  03 02 ad aa 00 00
col 2; len 6; (6):  03 02 ad aa 00 07
col 3; len 2; (2):  c8 38
----- end of leaf block dump -----
```

可以看出，行数变成了 7。每个键值的位图无法在一行中表示，这是因为数据的变化导致索引的行发生了裂变和重组。在原有的 8 行数据上增加 3 行数据，就导致索引发生如此大的变化，看来数据的变化对位图索引影响很大。

从上面的结果可以看出，在表数据发生变化时，位图索引也会发生大面积的变化，因此，在表上进行 DML 操作时，锁定记录的范围是一组记录，而不像普通 B 树索引一样只影响一行数据，这就会影响这张表上并发 DML 操作的性能。另外，经常进行 DML 操作的表，位图索引的碎片化也会很严重，索引访问的效率会有较明显的下降。

从上述两个原因，我们可以得出结论，位图索引不适合在 DML 操作十分频繁、并发量较大的表上使用。

从位图索引的结构特点来看，我们可以发现一个位图索引的优化方法。如果某张表相对静态，而且通过位图索引访问的频率很高，那么就可以考虑将表中数据按照键值排序组织，这样位图索引较小，访问效率最高。我们来看下面的示例：

```
SQL> select extent_id,file_id,block_id,blocks from dba_extents
where segment_name='IDX_TEST1';
 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
         0          1      88304          8
         1          1      88312          8
         2          1      88320          8
         3          1      88328          8
         4          1      88336          8
         5          1      88344          8
         6          1      88352          8
         7          1      88360          8
         8          1      88368          8
         9          1      88376          8
        10          1      88384          8
11 rows selected.

SQL> CREATE TABLE TEST1_1 AS SELECT * FROM TEST1 ORDER BY STATUS;
Table created.
SQL> CREATE BITMAP INDEX IDX_TEST1_1 ON TEST1_1(STATUS);
Index created.
SQL> SELECT EXTENT_ID,FILE_ID,BLOCK_ID ,BLOCKS FROM DBA_EXTENTS
WHERE SEGMENT_NAME='IDX_TEST1_1';

 EXTENT_ID    FILE_ID   BLOCK_ID     BLOCKS
---------- ---------- ---------- ----------
         0          1      91296          8
         1          1      91304          8
         2          1      91312          8
         3          1      91320          8
```

从上例可以看出，如果表数据按照位图索引列排序，那么位图索引就会比一般情况小很多，这个索引的访问开销当然也就很小。相对比较静态的数据，如果其他字段的范围扫描较少，而通过位图索引字段的访问较多，就可以考虑通过重组表数据来提高访问性能。

11.3 重建索引的作用

很多 DBA 在从事了多年维护工作后，依然不知道索引是需要定期维护的。实际上，索引比表更容易产生碎片，这是因为随着表数据的增加、删除和修改，索引在不停地重组。比如，如果要批量删除 1 万条记录，在删除过程中，没有提交之前，这些索引记录是不能被重用的，此时如果有新记录插入，那么就要分配其他的索引记录项，如果当前无法分配足够的记录项，就只能分

裂叶节点，以获取新的空间。这样的情形在日复一日地发生，因此索引会变得越来越臃肿，索引访问的效率也会逐渐下降。受影响最大的是范围扫描，因为随着索引碎片的增加，所需扫描的叶块数量也会有所增加。

老白曾经优化过一个银行系统，系统上线后 4 年都没有做过索引整理，有些索引已经比表都大了。在进行了一次索引重建后，第 2 天平均业务端到端响应时间提升了 30%多，这可是很多优化项目梦寐以求的效果啊。在 2011 年底老白参与的一个优化项目中，数据库并不大，但所有表和索引的扩展总容量却达到了 700 GB，经过索引重建后，扩展减少了 50 GB 多的空间。表 11-1 是老白最近一个优化项目中的实际案例。

表　11-1

索　引　名	表　　名	索引大小	表　大　小	占　比
PK_S_LINE_TG_RELA_ID	S_LINE_TG_RELA	2668	256	10
LOC_A_RCVED_PL_FLOW_R02_NO	A_RCVED_PL_FLOW	7770812	15142008	0.51
IDX_ARC_A_RCVBL_PL_FLOW_ACCT_N	ARC_A_RCVBL_PL_FLOW	7088100	13452312	0.52
LOC_G_TG_PQ_ORG_NO	G_TG_PQ	2842968	4562528	0.62
IDX_E_CONSPRC_SNAP_PRC_ID	E_CONSPRC_SNAP	1713552	268968	6.37
IDX_TACTIC_SP_ID	E_CONSPRC_ TACTIC_SNAP	1164536	223352	5.21
LOC_P_MID_RCVEDCOMP_RCVSECT	P_MID_RCVEDCOMP	1130192	1493840	0.75
IDX_A_PC_TRAN	A_PC_TRAN	1096272	629980	1.74
LOC_R_DATA_METER_ID	R_DATA	1016320	612960	1.65
LOC_E_CONS_SNAP_CONS_CONS_ID	E_CONS_SNAP	614056	428072	1.43
IDX_R_ENTITY_MAINT_APP_NO	R_ENTITY_MAINT	344192	294912	1.16
IDX_JK_SYCJDL_CONS_NOORGNO	JK_SYCJDL	318976	266624	1.19
PK_S_MP_METER_RELA_SCHM	S_MP_METER_RELA_SCHM	36864	4480	8.22

可以看到，某些索引的段大小甚至超过了基表的 10 倍。在这种情况下，对索引进行重建可以达到很好的优化效果。从最近几天整理的一些索引来看，平均每秒的事务数已经从 55 上升到了 96，而目前仅仅完成了一部分表的索引重建。

在本节的最后，老白将献上一个自己编写的脚本，用于分析表和索引的段关系，并找出可能存在的碎片化较严重的索引，如代码清单 11-1 所示。

代码清单 11-1

```
col owner format a15 truncate;
col table_name format a30 truncate;
col index_name format a30 truncate;
select
  idx.owner owner,
  idx.table_name tablename,
  idx.index_name index_name,
  idx.blocks idx_blocks,
  tbl.blocks tbl_blocks,
  trunc(idx.blocks/tbl.blocks*100)/100 pct
```

```
from
  (select i.owner owner ,i.index_name index_name,
      SUM(S1.blocks) blocks,i.table_owner table_owner,
    i.table_name table_name
    from dba_segments s1,dba_indexes i
    where
     s1.owner=i.owner and s1.segment_name=i.index_name and
     i.owner not in
    ('SYS', 'OUTLN', 'SYSTEM','MGMT_VIEW','SYSMAN','DBSNMP','WMSYS','XDB',
        'DIP','GOLDENGATE','CTXSYS' )
    GROUP BY i.owner ,i.index_name ,i.table_owner , i.table_name  ) idx,
  (select  t.owner owner ,t.table_name table_name,SUM(s2.blocks) blocks
    from dba_segments s2,dba_tables t
    where
     s2.owner=t.owner and s2.segment_name=t.table_name and
     t.owner
      not in
       ('SYS', 'OUTLN','SYSTEM','MGMT_VIEW','SYSMAN','DBSNMP','WMSYS','XDB',
           'DIP','GOLDENGATE','CTXSYS' )
     GROUP BY  T.OWNER,T.TABLE_NAME
     ) tbl
where
   idx.table_owner=tbl.owner and
   idx.table_name=tbl.table_name and
   (idx.blocks/tbl.blocks)>0.5 and
   idx.blocks>200
   order by 4;
```

这个脚本中有两个参数：一个是二者比较的阈值，这里老白使用了 0.5，实际上超过 0.1 就基本都是需要我们去关注的；另外一个是索引的大小，本例只分析超过 200 块的索引，更小的索引不在考虑范围内。实际上，这个脚本找出的索引并不一定都是有问题的，由于有些索引包含了表的主要字段，因此超过表的大小也是可能的。从中找出存在疑问的索引进行分析，而不要拘泥于这个脚本。

在线重建索引是一种高风险的操作吗

以老白的经历来说，在线重建索引是一件高风险的事情。在几年前的一次系统割接工作中，有一个操作员负责重建索引，他在提交了一个脚本后，就悠闲地坐在电脑桌前喝起了茶。有人从旁边经过时不小心碰了他一下，水洒到了笔记本电脑上，他本能地关闭电脑。正在他手忙脚乱地处理自己的事情时，旁边的业务系统突然开始出现大量的 ORA-00600: internal error code, arguments: [kkdlfjou_1]错误。经过检查发现，索引重建失败导致了索引损坏。此时重建或者删除都无法成功，重建这个索引，系统就会出现下面的错误：

```
ORA-08104: this index object 50528 is being online built or rebuilt

 COMMAND:
 alter table prod04.idx_acct_list rebuild online;
```

这是由于索引重建机制造成的，在线重建索引是基于日志表的。当开始在线重建索引时，系

统会创建一个临时日志表，这张表被用于存放索引重建期间产生的日志信息。同时在 IND$中，这个索引的 FLAG 字段上会被设置 REBUILD 标识（512），当索引信息变更时，会把变更信息存入日志表。而在线重建如果失败了，这个日志表和数据字典中的状态位都需要靠 SMON 来清理。要清理这个日志表，需要先锁定这张表，由于该表的数据一直在变化，SMON 无法对其进行锁定，因此就不能及时清理。如果索引所在的表变更十分频繁，那么这项清理工作就有可能会持续几天甚至几周才能完成。

鉴于在线重建索引是高风险的操作，因此老白建议进行该操作时要注意以下几点：

- 所有操作都写成脚本在后台以 nohup 方式运行；
- 准备好应急预案，一旦出现问题，马上采取措施；
- 对于高可用性要求的系统，不要让此操作在无人值守的情况下进行。

11.4　索引使用的“三大纪律八项注意”

通过前面几节的讨论，大家应该已经明白了索引的基本原理，也了解到了索引设计的一些基本要素。如果我们能根据每个 SQL 的特点来设计相应的索引，那么所有的 SQL 都可以通过最为适当的索引去访问数据，从而达到最佳的效果。不过这种设想往往是很难实现的，因为这样需要为每张表都设计数量庞大的索引，系统维护索引的成本就会变得十分高昂。那么我们应该如何来设计索引呢？

俗话说“花无百日红”，我们无法总是获得最佳的效果。因此在处理问题时，应该重点解决主要矛盾，索引设计也是如此。在设计时，我们必须为执行最为频繁、对系统性能影响较大的 SQL 设计成本开销最小的索引，而一些次要的 SQL，可以和其他 SQL 共用索引，这些索引可能并不是最优的，不过也能够满足目前应用的需求。而那些不是很常见的 SQL，其重要程度并不高，因此是否为它们设计索引，就应该视具体情况而定了。老白总结了一些索引设计方面的经验，并将它们归纳为“三大纪律八项注意”，在这里和大家一起分享。

三大纪律

第一，一切服从应用需要。在一张表上创建多少索引，创建什么样的索引，并无一定之规。不能说一张表上有了 7 个索引，就不能再创建第 8 个索引了。设计索引时，应该从应用的角度出发，不可轻易相信某些专家，只有应用开发者才对自己的应用最了解，请一个不了解具体情况的专家来设计索引并不一定合适，除非专家能够认真分析你的应用，不过这样的代价就相当高了。

第二，除非有很大的把握，否则不要轻易删除索引。在一套运行了很长时间的生产系统上，不经过严格论证就随意删除索引，可能会导致严重的后果。我在 10 年前曾碰到过一个案例，那一次在给客户做巡检时我发现了大量无用的索引，通过 MONITOR 工具观察了一个星期后，最终我将那些没有使用过的索引全部删除了，清理共 2000 多个。清理完成后某些应用的性能确实有所提高，而且系统也一直没有出现问题。但过了几个月，元旦那天，我接到了那个 DBA 的电话，

他哭笑不得地对我说：“老白，你把我害惨了，每年元旦早上我们老板都要看一个重要的数据，这回没出来，经过检查，少了个索引，就是你上回删除的。”从那以后，我再也不敢轻易动手清理索引了。

第三，要确保表和索引统计数据的一致性。如果表和索引的统计数据不匹配，出现执行计划错误的可能性就很大。如果相关的表和核心业务有关，那么将会造成严重的性能问题。前些天老白就碰到过一个这样的案例，在给一个客户做优化项目时，由于是在数据采集阶段，所以每天早上我都会打开 topas 工具观察系统的资源使用情况。有一天我发现 CPU 的 IDLE 几乎为 0 了，然后通过 vmstat 监控发现 r 队列达到了 50 左右，而这个指标平时只有 20 多。经过检查，我注意到有两条 INSERT ... SELECT 语句占用了近 20%的 CPU 资源。通过一个上午的分析，我发现这是由于统计数据的问题导致走错了索引。于是，我检查了表的 LAST_ANALYZED 字段，发现是 2011 年 12 月做的分析。难道是统计数据过旧导致了问题？这似乎也解释不通，因为之前一直是很正常的。后来我和客户的 DBA 进行了沟通，终于找出了问题的原因。原用户重建了一个索引，同时对索引做了分析，从而使索引和表的分析数据出现了不一致，进而导致了这个问题。在我们对这张表重新进行了分析后，这个问题就解决了。

八项注意

第一项：索引必须是有用的。不使用的索引不但会额外增加 DML 操作的成本，而且可能导致执行计划出现错误。因此必须从系统中清除这类索引。清除工作最好在系统上线前或者上线初期完成。在系统运行了很长时间后，即使监控到某个索引没有使用，也不能轻易地清除它。

第二项：尽可能让一个索引为更多的 SQL 服务，设计合理的复合索引是十分关键的。复合索引可以为单个过滤条件服务，也可以为组合过滤条件服务。我们无法为每种 WHERE 组合都设计独立的索引，因此统筹索引设计的关键是复合索引。索引设计者一定要把握好这一点。

第三项：尽可能将复合索引中选择性强的字段放在前面，这样可以减少索引范围扫描的成本。但在有些情况下，我们需要多些斟酌。比如，ID 和 CREATE_TIME 这两个字段，一个是主键，一个是创建时间，按照一般的情况，创建复合索引时，需要将 ID 放在前面，不过如果以 CREATE_TIME 为查询条件的 SQL 比较多，而 ID 的使用较少，那么我们在设计这个复合索引时，就要将 CREATE_TIME 放在前面。

第四项：在索引设计时要统筹考虑。比如，一张表上的访问过滤条件包括 PRODUCT_TYPE + PRODUCT_CLASSES 和 PRODUCT_TYPE + UPDATE_TIME 两种组合方式，我们可以创建两个索引，也可以考虑创建 PRODUCT_TYPE + PRODUCT_CLASSES + CREATE_TIME 的复合索引来替代两个索引的方案。

第五项：一定要根据实际情况选择适当的索引类别。位图索引和 B 树索引的使用场合是不同的，位图索引存储的是字段值的位图，因此修改某个索引字段的值时，系统维护索引的方式也和 B 树索引不同，维护时可能会同时锁定多条记录。基于位图索引的这一特点，在某张表针对索引字段的 UPDATE 操作或者某张表的 INSERT/DELETE 操作比较频繁的情况下，是不适合使用位

图索引的，否则很容易出现死锁。但也不要对位图索引存在偏见，有些 DBA 认为位图索引不适合在 OLTP 系统中使用，这也是错误的观点。只要使用得当，不出现死锁增加或者锁冲突增加的现象，在 OLTP 系统中使用位图索引也是可行的。

第六项：要注意函数索引的使用。绝大多数开发人员对函数索引了解不多，也很少在系统中使用，有些 DBA 还会故意夸大函数索引的维护开销。实际上，函数索引也是一种 B 树索引，不同的只是索引的键值是通过函数计算出来的。做过开发的人都很清楚，在表的字段上使用函数或者表达式是很难避免的，在某些情况下，去除表字段上的函数代价相当高。这时，函数索引就是最佳的选择。事实上，函数索引的维护成本并没有我们想象的那么高，和普通索引相比，函数索引的维护开销增加的部分就是函数计算的部分，在大多数情况下，这些成本都是可以承受的。

第七项：尽可能合并类似的索引。在一个大型系统中，由于表的查询条件在不断地变化，我们很可能会创建很多类似的索引。老白在以往的工作中就经常会碰到这样的情况，一套系统中存在 A+B+C 和 A+C+B 两个类似的索引，或者存在 A+C+E 和 A+B+E 这样的索引。实际上，这样的索引设计是很不好的，不仅增加了额外的开销，更大的危害是很容易引起 SQL 执行计划的不稳定，因此对这些索引进行适当的合并整理是十分必要的。

第八项：有时候在索引中增加部分额外字段可以起到很好的作用。对于一些查询量很大的大表，如果存在一条 SQL：

```
SELECT A FROM TAB1 WHERE B=:1 AND C=:2
```

针对上述情况，DBA 一般会创建一个 B+C 的索引，但实际上，如果创建一个 B+C+A 的索引，就可以避免对表的扫描，通过索引扫描就可以获得所需要的全部数据。

最后要提醒大家的是，索引是需要维护的，并不是创建后就万事大吉了。定期对索引进行评估和维护是十分必要的，通过定期检查 DBA_INDEXES 中相关统计数据的变化，我们可以粗略地了解索引的变化。比如索引的大小、B-LEVEL、CLUSTER FACTOR 等，这些都应该是我们关心的数据，在索引刚被创建或者重建的时候，应该保存这些数据，并定期检查一些关键索引的统计信息。如果发现这些数值变化较大，就需要对索引进行相关的维护，否则索引访问的效率会大幅度降低。

11.5　案例——索引危机

本节介绍一个索引优化的案例，通过这个案例，读者可以学习到如何分析、设计索引。另外，这个案例也介绍了一种非常规的索引优化方法，实际上在很多场合都可以使用此方法。

4 月 1 日　无法修改应用的项目

刚才在会议室和郭工一起讨论了这个项目，最近一两个月，郭工他们这套系统的 CPU 使用率增加了 20%左右，而白天业务高峰时，基本上都在 90%～100%，IDLE 基本上都是 0。郭工感

觉到系统的风险很大，于是向领导做了汇报，李总就急忙叫我过来帮他们分析分析。

听郭工介绍，这套系统 2 周后将迎来每个月最繁忙的时段，从目前的情况来看，这个 12 个 CPU 组成的 24 核的系统，r 队列已经达到了 40 多，如果不做一些优化，可能很难安全度过几天后的业务高峰。想要大幅度降低 CPU 的使用率，最好的办法就是优化应用。

于是，郭工把开发商负责维护系统的组长也叫到了会议室，商讨修改应用的事情。开发商一听要修改 SQL，头马上摇得跟拨浪鼓似的。在两周时间内完成 SQL 修改工作，对于开发商来说，确实难度不小。经过协商，开发商认为索引的修改和 PL/SQL 存储过程的优化他们能够马上配合我们进行，但修改程序的变更，他们必须上报给公司，走公司的变更流程，在两周内完成这项工作基本是不可能的。看样子开发商是指望不上了，于是我决定把优化的重点放在索引的调整上。如果能优化几个 BUFFER GET 较大的 SQL，让 CPU 使用率下降 10% ~ 15%，基本就能够完成任务了。

我采集了一下 10 点到 11 点的 STATSPACK 报告:

```
Cache Sizes (end)
~~~~~~~~~~~~~~~~~
               Buffer Cache:     8,000M      Std Block Size:         8K
           Shared Pool Size:     3,200M          Log Buffer:    10,240K

Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                  Redo size:          2,596,262.30             60,116.45
              Logical reads:            572,843.36             13,264.19
              Block changes:             22,523.53                521.53
             Physical reads:              2,637.11                 61.06
            Physical writes:                672.75                 15.58
                 User calls:             21,920.49                507.57
                     Parses:              6,339.75                146.80
                Hard parses:                310.56                  7.19
                      Sorts:              1,714.43                 39.70
                     Logons:                  0.44                  0.01
                   Executes:              6,434.11                148.98
               Transactions:                 43.19

  % Blocks changed per Read:    3.93    Recursive Call %:      9.04
 Rollback per transaction %:    4.01       Rows per Sort:     48.64

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Buffer Nowait %:   99.98       Redo NoWait %:     99.99
            Buffer  Hit   %:   99.60    In-memory Sort %:    100.00
            Library Hit   %:   97.61        Soft Parse %:     95.10
         Execute to Parse %:    1.47         Latch Hit %:     99.36
Parse CPU to Parse Elapsd %:   62.06     % Non-Parse CPU:     90.83
```

```
 Shared Pool Statistics        Begin    End
                              ------  ------
             Memory Usage %:  100.00  100.00
    % SQL with executions>1:   30.49   31.15
  % Memory for SQL w/exec>1:   34.11   34.20

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                     % Total
Event                                               Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file sequential read                         5,023,080      75,306    41.99
CPU time                                                       74,925    41.77
global cache cr request                         4,851,421       6,662     3.71
db file scattered read                            181,344       3,582     2.00
latch free                                      4,553,880       3,572     1.99
-------------------------------------------------------------------------------
```

从报告上看，CPU time 在所有的事件中排在第二位，基本上和 db file sequential read 事件差不多。另外，Non-Parse CPU 只有 90%左右，说明解析消耗的 CPU 也不少。但由于只有两三天的时间，这部分优化工作就暂时不考虑了。从等待事件上看：

```
                                                                   Avg
                                                     Total Wait   wait    Waits
Event                           Waits     Timeouts    Time (s)    (ms)     /txn
---------------------------- ------------ ---------- ---------- ------ --------
db file sequential read         5,023,080          0     75,306     15     32.3
global cache cr request         4,851,421      4,394      6,662      1     31.2
db file scattered read            181,344          0      3,582     20      1.2
```

I/O 性能不算太好，不过也还算凑合，不会有太大的性能影响。这次优化重点放在索引调整上，因此需要考虑的其他方面的问题较少，只要确保索引调整后，TOP SQL 的执行计划不会变坏就行了。基于这样的考虑，整个优化工作就变得很简单了。首先挑出需要优化的 SQL，然后分析是否可以通过调整索引进行优化，如果可以，就做出一个索引调整方案，由开发商来实施即可。一些较小的索引调整可以立即进行，但如果索引的大小超过 200 MB，就需要在中午或者晚上业务较少的时间进行。

我在 STATSPACK 报告中查找了逻辑读较大的 SQL，排在第一位的是一条一小时执行了 50 万次的 SQL，每次开销为 1000 多个 BUFFER GET。在查看了执行计划后，我发现它通过一个索引做范围扫描，看样子也没有多大的问题，只是执行频率较高而已。排在第二位的是一条很简单的 SQL：

```
SELECT ORDER_ID ,
    NVL(PAYMENT_ID, -1) PAYMENT_ID,
    ACCT_ID,
    SERV_ID,
    ACTION,
    STATE ,
    TO_CHAR(CREATED_DATE,'YYYYMMDDHH24MISS'),
    TO_CHAR(STATE_DATE,'YYYYMMDDHH24MISS')   FROM A_XXX
    WHERE NVL(PAYMENT_ID,-1)>=:lPaymentID    AND STATE=:sState  ORDER BY
    NVL(PAYMENT_ID, -1), CREATED_DATE
```

这条 SQL 每次访问的 BUFFER GET 只有不到 2500 个，不过访问的次数不少，有五六万次。实际上，这条 SQL 的写法是有问题的，PAYMENT_ID 字段上是有索引的，不过由于使用了 NVL 函数，导致这个索引无法使用。其实 PAYMENT_ID 肯定是一个大于 0 的字段，由于在该字段上有一个“>”过滤条件，因此满足该条件的记录 PAYMENT_ID 肯定不为空，所以如果把 NVL 函数去掉，改为：

```
SELECT ORDER_ID ,
       NVL(PAYMENT_ID, -1) PAYMENT_ID,
       ACCT_ID,
       SERV_ID,
       ACTION,
       STATE ,
       TO_CHAR(CREATED_DATE,'YYYYMMDDHH24MISS'),
       TO_CHAR(STATE_DATE,'YYYYMMDDHH24MISS')   FROM A_XXX
       WHERE PAYMENT_ID>=:lPaymentID    AND STATE=:sState  ORDER BY
       PAYMENT_ID, CREATED_DATE
```

从语义上来看，这两条 SQL 是完全等价的。由于 PAYMENT_ID 字段是十分常用的，因此必须保留 PAYMENT_ID 上的索引，所以优化这条 SQL 的最佳方案是修改程序，去除 PAYMENT_ID 上的 NVL 函数。在和开发商沟通后，他们也认为这是开发人员的一个失误，这个 NVL 函数是可以去掉的，不过修改程序的审批和测试流程十分繁琐，短期内很难实施，最好能使用其他解决方案。如果要找其他解决方案，就只能考虑在 NVL(PAYMENT_ID,-1)上创建函数索引了。根据开发人员的介绍，这个 SQL 是查询 PAYMENT_ID 大于某个值的未处理过的记录的数量，STATE 的值只有五六个，选择性一般。开发人员帮我找到一个 PAYMENT_ID，我通过 SELECT COUNT(*) FROM A_XXX WHERE PAYMENT_ID>:PID 语句查询了一下，满足大于 PAYMENT_ID 的记录大约占整个表的 1/15。于是我又加上 STATE 的过滤条件查询了一次，查出的记录总数占整个表的 1/60。在这种情况下，创建 NVL(PAYMENT_ID)+STATE 的索引，效果比创建 PAYMENT_ID 的单列索引要好得多。A_XXX 表的总记录数为 200 多万条，因此在业务空闲期创建索引对系统影响不大。于是，我建议中午就在这张表上创建这个索引。

中午时，开发商创建了这个复合索引。我午饭后回到办公室是下午 1:30，查看了系统的负载情况，CPU 使用率只有 50%左右。我知道这肯定不是刚才那个索引的功劳，可能是中午很多营业厅都没什么业务的缘故。我做了一次收集级别为 6 的 STATSPACK 分析，从生成的 STATSPACK 报告中可以看出，这条 SQL 在半小时里执行了不到两万次，平均每次执行的 BUFFER GET 的数量是 1500 左右，比刚才有了明显的下降，应该是执行计划发生了改变。为了确认这一点，我又使用 SPREPSQL 生成了这条 SQL 的报告，在 SQL 报告中，我明确地看到 SQL 执行时使用了新建的索引。

下午 3 点到 5 点是业务高峰期，在 2 点半之后，系统负载就在不断地上升，不过 CPU 使用率一直徘徊在 80%左右。我心中难免有些喜悦，难道刚才那个索引居然取得了这么好的效果？

3 点 10 分左右，CPU 使用率再次突破了 90%，几分钟后，达到了 95%，到 3 点 30 分左右，IDLE 终于消失了。虽然我有点失望，不过这也在预料之中，想通过一两个索引就达到优化的目的，是不太现实的。在这套系统中，我还有很多的工作要做。

4 月 2 日　索引范围扫描的优化方法

昨天下午又优化了几个 SQL，优化后都有了明显的性能提升，不过从今天上午 10 点多观察到的 CPU 使用率来看，好像变化不大。由于排队效应的存在，简单的减法并不能完全解决问题。解决了以前排在前面的几个 SQL 的问题后，又出现了一些新的 SQL，其中一条 SQL 每次执行的 BUFFER GET 数量高达 3 万多：

```
select a.filename, b.source_id, b.source_path, b.localnet_abbr,
b.file_fmt,b.is_txt_fmt, a.deal_flag
from SCH_XXX a, BBB b
where (a.deal_flag='W' or a.deal_flag='B') and
 a.validflag='Y' and b.pipe_id=:szPipeId and a.source_id=b.source_id
```

具体的执行计划如下：

```
--------------------------------------------------------------------------------
| Operation                        | PHV/Object Name     |  Rows | Bytes|   Cost |
--------------------------------------------------------------------------------
|SELECT STATEMENT                  |----- 1104374539 ----|       |      |   3486 |
|HASH JOIN                         |                     |    79K|    7M|   3486 |
| TABLE ACCESS BY INDEX ROWID      |BBBBB                |     2 |    98|      2 |
| INDEX RANGE SCAN                 |FK_SOURCE_WORKFLOW   |     1 |      |      1 |
| TABLE ACCESS FULL                |SCH_XXXXX            |     1M|   61M|   3474 |
--------------------------------------------------------------------------------
```

从这个执行计划来看，对 SCH_XXXX 的扫描是开销最大的，这张表中扫描的记录数为 100 万（1M）。通过 SPREPSQL 报告，我们看到：

```
                     Statement Total      Per Execute
                     ---------------  ---------------
        Buffer Gets:         626,984         39,186.5
         Disk Reads:         425,693         26,605.8
     Rows processed:              46              2.9
     CPU Time(s/ms):              50          3,113.1
 Elapsed Time(s/ms):             988         61,779.7
              Sorts:               0               .0
        Parse Calls:              16              1.0
      Invalidations:               0
      Version count:               2
    Sharable Mem(K):              29
         Executions:              16
```

平均每次执行返回的记录数是 2.9 条，也就是说真正符合条件的记录只有区区 3 条左右。这和执行计划中的 79K 相差甚远，于是我检查了这张表，这张表总计有 200 万条记录。在这张表上，存在两个过滤条件(a.deal_flag='W' or a.deal_flag='B')和 a.validflag='Y'。我首先分析了一下 validflag：

```
Select validflag,count(*) from sch_xxxx group by validflag;
```

发现表中的大多数记录都是 validflag='Y'的记录，这个字段的选择性是很差的，接着，我又检查了 deal_flag，发现绝大多数记录都是 deal_flag='Y'的，其他的记录非常少，有时甚至是 0。

这是一个典型的字段倾斜案例，凑巧的是，在这个 SQL 中并没有使用绑定变量，因此在 deal_flag 上创建一个索引，并且分析一下柱状图就可以解决问题。经过和客户的协商，由于这张表只有 200 万条记录，因此我们决定立即创建索引，并做表分析，于是我马上执行了下面的脚本：

```
CREATE INDEX SHCJ.IDX_DEALFLAG ON SHCJ.SCH_XXXX(DEAL_FLAG) ;
Exec DBMS_STATS.GATHER_TABLE_STATS(ownname=> 'SHCJ', tabname=> 'SCH_XXXX',
estimate_percent=>30, method_opt=>'for all indexed columns size
skewonly',cascade=>true, degree=>2);
```

索引创建后，我再次分析了表和索引，发现表对索引字段中的倾斜字段分析了柱状图。这里可以将 Method_opt 设置为'for all columns size auto'，这是 10*g* 版本的默认值，而 9*i* 版本的默认值是'for all columns size 1'，也就是不分析柱状图。

上述操作完成后，我马上对这个 SQL 进行了分析，发现 BUFFER GET 的数量从近 4 万减少到了 10.7，这个 SQL 产生的物理读基本也没有了。这说明索引起到了很好的作用。

从上午业务高峰期的情况来看，虽然我又调整了三四个 SQL，不过总体的效果并没有很明显的提升。在上午 10 点到 11 点 30 分之间，系统的 CPU 使用率仍然处于 95%～100%的高位，虽然平均事务响应时间有了 20%左右的提升，但还是很难彻底解决问题。在 STATSPACK 报告中，已经找不到明显的可以通过索引调整进行优化的 SQL 了，如何解决 CPU 负荷过高的问题，好像还没有半点眉目。这套系统的 SQL 开销比较平均，除了排在 BUFFER GET 第一位的那条 SQL 占了整个开销的 37%外，其他的 SQL 占整个系统 BUFFER GET 的比例都不高，最多的也只是在 4%左右。调整这些 SQL，只能取得有限的效果。看来想要走捷径，就必须优化这个排名第一的 SQL。于是我再次查看了这个 SQL，它涉及两张表：稍大的一张是 T_PRODUCT_INFO，这张表的 WHERE 条件中有一个 PRODUCT_TYPE=:P1 AND PRODUCT_CLASS=:P2 过滤条件；另外一张是小表，大概只有 20 多条记录。执行计划是，在扫描完两张表后，对结果集做散列连接（HASH JOIN）。在 T_PRODUCT_INFO 表中，存在一个复合索引（PRODUCT_TYPE,PRODUCT_CLASS），执行计划也是通过这个索引进行范围扫描的。从执行计划来看，并没有问题，平均每次返回结果的记录数不过 60 多条，而 T_PRODUCT_INFO 进行索引范围扫描找到的记录接近 700 条。T_PRODUCT_INFO 表的总记录数为 800 万，从一个包含 800 万条记录的表中，扫描出 700 条记录，这个索引的效率还是比较高的。可是为什么这样一个简单的 SQL 会产生 1500 多个 BUFFER GET 呢？我马上联想到了聚簇因子（cluster factor），于是立即查看 DBA_INDEXES 视图，果然，这个索引的聚簇因子接近记录的数量。如果聚簇因子较高，在通过索引范围扫描访问表时，由于每个索引键值更多地指向新数据块，因此会产生更多的物理 I/O，并导致 BUFFER GET 的数量增加。

我马上找到郭工，了解这张表的用途。这张表是存放产品资料的，客户做售后服务时会通过 PRODUCT_TYPE 和 PRODUCT_CLASS 找到符合条件的备件，然后再根据库存情况以及本次服务的其他限定条件，找出满足条件的备件。这张表的数据是逐步添加的，随着新产品及其备件的出现，表数据也越来越多。以前这张表只有一两百万条记录，随着这几年的积累，记录数已经接近 1000 万了。我想了一下，这张表的记录的加入，肯定不会按照 PRODUCT_TYPE 和

PRODUCT_CLASS 进行排序，因此索引的聚簇因子较大也就不足为奇了。于是我问郭工，对这张表的访问是不是大多数都通过 PRODUCT_TYPE 和 PRODUCT_CLASS 进行，郭工想了想，说："是的，绝大多数访问都是根据这两个字段的条件，另外一种就是根据 PRODUCT_ID 直接定位产品。" PRODUCT_ID 是这张表的主键，既然主要的访问方式只有这两种，那么我倒是想出了一个优化方案。

由于记录的插入顺序没有按照 PRODUCT_TYPE 和 PRODUCT_CLASS 排序，因此索引范围扫描的成本较高。如果我们能够对这张表进行重组，使之按照这两个字段排序，那么范围扫描的成本就可以得到有效的减少。采用如下的方法可以完成这个优化：

(1) 将原表重命名为 T_PRODUCT_INFO_OLD；

(2) 使用下列语句创建新表，并按照 PRODUCT_TYPE 和 PRODUCT_CLASS 排序；

Create table t_product_info as select * from t_product_info old order by product_type,product_class；

(3) 创建相关索引及约束关系；

(4) 分析表和索引。

于是，我们决定今晚就实施优化操作。郭工听我说了半天聚簇因子，也没明白，便说："老白，技术的问题我就不管了，我帮你申请今晚停机的事情。这套系统晚上就只有一些统计报表，所以在 19 点到 21 点之间都是可以停库的。如果你的操作能够在 2 个小时内完成，停机申请就没有任何问题，否则我就需要和业务部门协商了。" 我想了想，这张表也就几百兆字节的大小，1 个小时肯定能解决问题了。

4 月 3 日　效果不错

昨天晚上重建了 T_PRODUCT_INFO 表，操作过程还是比较顺利的，尽管客户申请了一个小时的停机窗口，但不到半小时，整个操作就完成了。原本计划 8 点停机，但因为有一个很重要的日报表要跑，最后拖到了 10 点多才开始。在这段时间里，我又对 BUFFER GET 和 BUFFER BUSY WAITS 比较大的几个索引进行了一些分析，发现了一张表，表的大小只有 200 MB，但索引的大小却接近 300 MB，索引碎片十分严重。于是我对一些访问比较频繁的表上的索引进行了一番检查，发现这种现象还是比较普遍的。在和郭工商量后，我们决定对这些索引进行重建。

由于停机时间比较充裕，做完表重建后还有半个多小时，我又重建了几张比较重要的小表的索引。那个接近 300 MB 的索引，重建后只有 60 MB 多了。由于这套系统自从上线后基本没做过重建，因此存在很多碎片化比较严重的索引。我和郭工商定在应用恢复后，对小于 1 GB 的访问比较频繁的表上所有的索引进行一次在线重建。这些索引总计 20 多个，最大的有 1 GB 多。为了保险起见，我建议在控制台上进行索引的在线重建工作，以前曾经碰到过由于网络中断导致重建失败的情况，正在进行的在线重建操作出错，正在被重建的索引处于非正常状态，无法删除，也无法重新继续重建。最后只能通过 Metalink 上提供的方法，手工删除日志表，并且修改了 ind$ 中索引的状态属性，才解决了问题。

索引重建后，我又对相关的表进行了 CASCADE=>TRUE 的分析，完成时已经是凌晨两点多

了。开发商用预先准备好的十几条关键业务相关的 SQL 做了一次测试，确认了执行计划都是正确的。完成这些工作后，郭工还是不太放心，劝我留下，万一明天早上有事，可以马上处理。这也是我经常遇到的，于是我很痛快地答应了。

第二天早上我没有接到郭工的电话，看样子昨晚的调整并没有产生负面影响。午饭后，我来到办公室查看系统的情况，CPU 使用率只有 60%左右，虽然现在是业务空闲时段，不过前几天这个时间段的 CPU 使用率都在 80%～90%的高位，这说明昨天的工作还是有一定效果的。听小马说，上午 CPU 使用率也一直比较稳定，在 80%～85%之间，很少有达到 90%的情况。我查看了 OSW 的监控数据，确实，今天上午的 CPU 使用率都在 80%左右，只有在 10 点半左右出现了几个 90%～95%之间的采样点。我马上生成了一个 STATSPACK 报告，查看那条访问 T_PRODUCT_INFO 的 SQL 的开销情况，每秒的逻辑读和物理读都下降了 40%多，执行时间少了 50%多。昨天的表重组起到了不错的效果，虽然这条 SQL 的 BUFFER GET 还是排在前五，不过占整个系统 BUFFER GET 的比例已经下降到了 7%左右。这样看来，如果下午能够保持这种状态，本次优化工作就可以告一段落了。

下午 3 点多，郭工回到办公室，我们约定下午 4 点讨论优化报告，如果没有问题，大家就签署一个工作日志，然后结束本次优化工作。本来商定的是我和郭工、小马一起开个小会，谁料 3 点半时郭工突然通知我，他们业务中心的刘主任也要参加这次会议。考虑到刘主任是负责业务的，对技术并不在行，如果仅仅讨论这份不到 10 页的优化工作报告，效果肯定不好，于是我做了一份 PPT，简单介绍了本次优化的背景、目的、实施方案以及达到的效果，为了更直观地展示效果，还使用柱状图和折线图比对了优化前、后系统的资源消耗和性能的差异，从图表上看，这次优化做得还算成功。

总结会上，我向大家介绍了优化的全过程和取得的丰硕成果，刘主任听后也很满意。他在开会前已经了解过营业员对系统的反馈，早就听说他是十分严谨的人，从这件事来看确实如此。还好我准备了 PPT，列出了具体的数字，否则会议的效果肯定大打折扣。本以为介绍完这些情况会议就结束了，没想到我正等着刘主任的客套话时，他突然提出了一个问题："老白，从你的报告里我感觉在这次优化中，起到最重要作用的就是表的重建，是不是？"

我想都没想就回答说是，这次优化最关键的确实就是表重组，通过重组减少了索引范围扫描的开销。还没等我继续解释，刘主任又接着问到："这回的问题是解决了，如果过一段时间，这张表的性能又下降了，我们的系统是不是又会出现类似的问题？"

我回答说："是的，所以过一段时间这张表还需要再次重建，否则还是会有些影响，不过这张表刚刚重建过，估计在半年之内问题不会太大。"

刘主任眉头一皱："这张表会越来越大，那么随着时间的推移，这张表重建的难度也将越来越大，有没有更好的办法能够确保重建工作在较短的时间内完成呢？重建工作会对业务造成影响，最好每次重建的停机时间都能够控制在 30 分钟以内。"

这个问题我确实没有考虑过，因为目前这张表只有不过 800 多万条记录。但是它的增长速度十分快，今年年底前记录数可能就会翻番，如果每次都去做全表重组，确实有点麻烦。于是我向郭工询问这张表的数据特点，郭工说，这张表的数据一般在插入后就很少会被修改，除非是某个

产品或者备件的状态发生了变化，比如某个产品退市了，会修改这个产品的状态位。原则上，这张表的数据是只增加不删除的。

根据郭工的描述，我认为通过分区表的方式就能够解决这个问题。如果我们用产品输入时间作为分区关键字，那么就可以根据产品输入的时间将表分为若干个分区，如果每个分区包含 6 个月的数据，那么大约会有 500 万条记录。如果我们将索引创建为本地索引，那么就可以每隔半年重组一个分区，每次重组的时间肯定不会超过半小时。这样的话，虽然索引范围扫描的开销要略大于现在的普通表的模式，但是能够很好地解决长期维护问题。

这个建议提出后，郭工感到有些茫然，他觉得使用本地索引后可能会影响范围扫描的性能，本来是扫描一个全局索引，而现在要对所有的分区进行扫描，这样是不是会增加 SQL 的开销，还不如把索引建成全局索引。

我解释道："如果建成全局索引，那么索引扫描的效率在分区前后的差异相当小，基本上可以忽略不计。但是每次重组分区时，全局索引都需要重建，随着表的数据量增加，索引重建的时间也会越来越长，这样就会影响停机时间。"

确实，这是一种两难的选择，但在便于维护和更好的性能这两方面，一般来说，选择便于维护会更好。维护不是简单的加、减、乘、除，理论上的最优选择往往不是实际工作中的最佳答案。

第 12 章

理解分区表

12

分区表技术的引入是 Oracle 在海量数据处理方面一项很大的进步，它已成为海量数据管理中十分重要的技术手段。通过分区表技术，我们可以把一张很大的表分为多个较小且可管理的分区，从而规避性能和维护方面的问题。很多 DBA 对分区表的了解并不深入，因此在使用分区表技术的过程中会遇到很多问题。

在本章中，老白将和大家一起从分区表最为本质的特性出发，学习如何使用、维护分区表，从而使分区表技术在优化中发挥最大的效能。

12.1 什么是分区表

分区表技术是 Oracle 8.0 推出的，到现在为止已经有十多年，不过直到近几年，才在中国真正被重视和使用。分区表的出现是 VLDB 系统发展的必然结果，随着数据量的增加，分区表的作用也日益凸显。实际上，早期的 VLDB 系统数据量并不大，只有 1 ~ 200 GB。最早有分区表业务需求的是电信和移动的计费系统，每个月几千万条话单的数据，如果放在一张表里，处理起来肯定会出现问题。通信行业集成商采用的解决方案是分表，将不同账期的数据放在不同的表中。分表虽然有效地解决了处理这些数据时可能出现的性能问题，但也给开发团队带来了另外一个问题，就是应用开发的复杂度问题。同一个业务的数据可能存放在多张名称类似的表中，用户需要在多张表中查询数据，这使得应用的复杂度大幅增加。

分区表技术有效地解决了这两方面的问题，既保证了数据的分区存放，确保每个分区可以独立访问，同时又可以对整个表中的各个分区进行透明访问，并且不需要对原来的应用程序进行调整。分区表技术是很适合在运营商的计费账务系统中使用的，但是国内早期的成功应用并不是出现在上述系统中。在 2000 年到 2006 年这段时间里，我做了不少大型的系统性能优化项目，在这些项目中无一例外地都使用了表分区的技术，通过将普通表转换为分区表或者调整分区表的分区方式来对系统进行优化。在实际应用过程中，这种调整在绝大多数情况下都取得了不错的效果。

近年来，越来越多的软件开发厂商开始在自己的系统中使用分区表技术，并从这种技术中获得了不错的效果。不过还是有一些用户对分区表技术仍然十分抵触，认为该技术增加了维护的复杂性。在接下来的内容中，我们就从多个角度来了解分区表。

分区表包含多种类型，比如范围分区、HASH 分区、复合分区、列表分区等，为了简化起见，我们先从使用最为普遍的范围分区说起。为了便于大家理解，这里先从创建一张范围分区的分区表开始讲述，如代码清单 12-1 所示。

代码清单 12-1

```
SQL> create table XJTPAR (
  2    id integer not null,
  3    txt varchar2(100),
  4    tdate date
  5    )
  6    PARTITION BY RANGE (ID)
  7  (
  8    PARTITION XJPART_01 VALUES LESS THAN (10000),
  9    PARTITION XJPART_02 VALUES LESS THAN (20000),
 10    PARTITION XJPART_03 VALUES LESS THAN (30000),
 11    PARTITION XJPART_04 VALUES LESS THAN (40000)
 12    );

Table created.
```

这样就创建了一张分区表 XJTPAR，这张表包含 4 个分区，分别是 XJPART_01、XJPART_02、XJPART_03 和 XJPART_04。首先来看一看，这些表和分区是否为独立的对象，我们可以从 DBA_OBJECTS 里查找：

```
SQL> col object_name format a10 trunc
SQL> col subobject_name format a12 trunc
SQL> set line 132
SQL> select object_id,object_name,subobject_name,object_type,data_object_id from
dba_objects where object_name='XJTPAR';

OBJECT_ID   OBJECT_NAM     SUBOBJECT_NA       OBJECT_TYPE         DATA_OBJECT_ID
---------- ------------- ---------------- -------------------- ------------------
124950       XJTPAR         XJPART_01        TABLE PARTITION         124950
124951       XJTPAR         XJPART_02        TABLE PARTITION         124951
124952       XJTPAR         XJPART_03        TABLE PARTITION         124952
124953       XJTPAR         XJPART_04        TABLE PARTITION         124953
124949       XJTPAR                          TABLE
```

从上述结果可以看出，分区表 XJTPAR 是一个对象，ID 是 124949，而每个分区则分别是一个子对象。实际上，每个分区都是一个独立的子对象。Oracle 的这种设计，既兼顾了其原有的对象概念，又使分区表的每个分区有了独立的对象资格，并且每个子分区都有自己独立的 DATA_OBJECT_ID，从而使每个分区有了独立的段。在这里，细心的朋友可能会发现一个问题，表分区的 DATA_OBJECT_ID 都是存在的，反而是 XJTPAR 的 DATA_OBJECT_ID 是空的。事实上，分区表的数据是存储在每个分区中的，表是不需要存储数据的，因此也就不需要为表自身分配段。我们可以通过 DBA_SEGMENTS 验证这个结论：

```
SQL> col segment_name format a10 trunc
SQL> col subsegment_name format a12 trunc
SQL> select segment_name,partition_name,segment_type,blocks from dba_segments where
```

```
segment_name ='XJTPAR';

SEGMENT_NA PARTITION_NAME                 SEGMENT_TYPE          BLOCKS
---------- ------------------------------ ------------------ ----------
XJTPAR     XJPART_01                      TABLE PARTITION            8
XJTPAR     XJPART_02                      TABLE PARTITION            8
XJTPAR     XJPART_03                      TABLE PARTITION            8
XJTPAR     XJPART_04                      TABLE PARTITION            8
```

我们可以看到，对于 XJTPAR 这张分区表，每个分区都有一个段，而表本身并没有段存在。

因此，我们可以大胆推断，每个表分区（TABLE PARTITION）本身都有独立的段头，这一点可以通过 DATA BLOCK DUMP 来进行验证。首先我们通过 DBA_EXTENTS 来查看这几个段的扩展情况：

```
SQL> select segment_name,partition_name,blocks,file_id,block_id from dba_extents
where segment_name ='XJTPAR';

SEGMENT_NA PARTITION_NAME                     BLOCKS    FILE_ID   BLOCK_ID
---------- ------------------------------ ---------- ---------- ----------
XJTPAR     XJPART_01                               8          4        521
XJTPAR     XJPART_02                               8          4        529
XJTPAR     XJPART_03                               8          4        537
XJTPAR     XJPART_04                               8          4        545
```

接下来，通过 DUMP XJPART_01 和 XJPART_02 的前几个块来验证我们的推断：

```
SQL> alter system dump datafile 4 block min 521 block max 524;

System altered.

SQL> alter system dump datafile 4 block min 529 block max 533;

System altered.
```

我们可以从 TRACE 文件中看到：

```
Start dump data blocks tsn: 4 file#: 4 minblk 521 maxblk 524
buffer tsn: 4 rdba: 0x01000209 (4/521)
scn: 0x0000.055f126c seq: 0x01 flg: 0x04 tail: 0x126c2001
frmt: 0x02 chkval: 0x7b74 type: 0x20=FIRST LEVEL BITMAP BLOCK
Hex dump of block: st=0, typ_found=1
...

buffer tsn: 4 rdba: 0x0100020a (4/522)
scn: 0x0000.055f126b seq: 0x01 flg: 0x04 tail: 0x126b2101
frmt: 0x02 chkval: 0x6964 type: 0x21=SECOND LEVEL BITMAP BLOCK
Hex dump of block: st=0, typ_found=1

...

buffer tsn: 4 rdba: 0x0100020b (4/523)
scn: 0x0000.055f126c seq: 0x01 flg: 0x04 tail: 0x126c2301
frmt: 0x02 chkval: 0x46c7 type: 0x23=PAGETABLE SEGMENT HEADER
```

```
Hex dump of block: st=0, typ_found=1

...

buffer tsn: 4 rdba: 0x0100020c (4/524)
scn: 0x0000.001d78df seq: 0x01 flg: 0x06 tail: 0x78df0601
frmt: 0x02 chkval: 0x55bc type: 0x06=trans data
Hex dump of block: st=0, typ_found=1
```

对于子分区 XJPART_01，其存储结构和普通表一样：首先是 1ST BMB，然后是 2ND BMB，第 3 个块是段头。XJPART_02 的结构也是类似的：

```
Start dump data blocks tsn: 4 file#: 4 minblk 529 maxblk 533
buffer tsn: 4 rdba: 0x01000211 (4/529)
scn: 0x0000.055f1287 seq: 0x01 flg: 0x04 tail: 0x12872001
frmt: 0x02 chkval: 0x7b75 type: 0x20=FIRST LEVEL BITMAP BLOCK

...

buffer tsn: 4 rdba: 0x01000212 (4/530)
scn: 0x0000.055f1286 seq: 0x01 flg: 0x04 tail: 0x12862101
frmt: 0x02 chkval: 0x697d type: 0x21=SECOND LEVEL BITMAP BLOCK

...

buffer tsn: 4 rdba: 0x01000213 (4/531)
scn: 0x0000.055f1287 seq: 0x01 flg: 0x04 tail: 0x12872301
frmt: 0x02 chkval: 0x46c6 type: 0x23=PAGETABLE SEGMENT HEADER
```

每个分区都具有独立的和表一样的存储结构，拥有独立的段头、位图以及高水位标志。这就决定了在大并发量插入数据时，分区表在段头、BMB 和高水位推进方面会有更大的优势。

另外一方面，由于每个子分区都拥有和普通表一样的段结构，因此分区表某个分区的扫描性能和由一张分区表划分而来的每张独立表的扫描性能是没有差异的。说到这里，可能有些朋友会有不同的看法，认为普通的分表访问数据时，不需要判断数据在哪张表中，而分区表则需要判断数据在哪个分区中，这样效率肯定会受到影响。事实上，扫描操作到底扫描哪个段，是在 SQL 分析时就确定了的，当然存在分区对象的 SQL 分析需要访问更多的数据字典，以确定执行计划，这一点是肯定的，不过这点差异我们可以暂时忽略不计。

从分区表的发展历史来说，Oracle 8.0 是最早推出分区表技术的版本，不过这个版本只支持范围分区。Oracle 8*i* 增加了 HASH 分区，同时开始支持子分区，另外，8*i* 版本还支持第一层为范围分区，第二层为 HASH 分区的复合分区表。Oracle 9.0.1 开始支持列表（LIST）分区，9.0.2 开始支持范围分区作为第一层，列表分区作为第二层的复合分区。Oracle 9.2 开始支持一种有些怪异却十分有用的混合模式分区——Range-List 分区。代码清单 12-2 是一个 Range-List 分区的示例（该示例来自于 MOS 文档 Range List Partitioning - Oracle 9.2 Enhancement [ID 209368.1]）。

代码清单 12-2

```
CREATE TABLE empdata
( empno number,
  ename varchar2(20),
  deptno number,
  continent varchar2(6),
  hiredate date,
  job varchar2(10),
  salary number)
PARTITION BY RANGE (deptno)
SUBPARTITION BY LIST (continent)
(PARTITION d1_emp VALUES LESS THAN (100)
 (SUBPARTITION d1_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d1_con2 VALUES ('AMER'),
  SUBPARTITION d1_con3 VALUES ('AFRICA'),
  SUBPARTITION d1_con4 VALUES ('EUROPE')),
 PARTITION d2_emp VALUES LESS THAN (maxvalue)
 (SUBPARTITION d2_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d2_con2 VALUES ('AMER'),
  SUBPARTITION d2_con3 VALUES ('AFRICA'),
  SUBPARTITION d2_con4 VALUES ('EUROPE'))
);
```

Oracle 10*g* 继续对分区表的功能进行扩展，最大的变化就是开始支持索引组织表的分区。代码清单 12-3 的示例同样来自于 MOS（Partitioning Enhancements in Oracle 10g [ID 276158.1]）。

代码清单 12-3

```
SQL> CREATE TABLE sales (
acct_no NUMBER(5),acct_name CHAR(30),amount_of_sale NUMBER(6),
week_no INTEGER,sale_details VARCHAR2(1000),
PRIMARY KEY (acct_no, acct_name, week_no))
ORGANIZATION INDEX INCLUDING week_no
OVERFLOW tablespace overflow_here
PARTITION BY LIST (week_no) (
PARTITION part1234 VALUES (1, 2, 3, 4) tablespace ts1,
PARTITION part5678 VALUES (5, 6, 7, 8) tablespace ts1,
PARTITION partdefault VALUES (DEFAULT) tablespace ts1);
```

Oracle 11*g* 在分区表技术方面又有了新的扩充，主要包括以下几个方面（这里仅对 11*g* 版本分区表的新特性做简单的描述，关于 11*g* 版本分区表更详细的信息请参考 MOS 文档 11*g* Partitioning Enhancements [ID 452447.1]）。

- 系统自动分区表。这种分区表和普通分区表不同，没有分区主键，用户可以通过在 INSERT 操作中指定分区来把数据插入到某个分区中。一般在 OLAP 系统中使用，OLTP 较少使用。这种分区的另外一个意义是打散数据，减少冲突，这个特性在 RAC 上有用武之地。
- 间隔分区（interval partitioning）。这个功能对于那些觉得维护分区表比较头痛的朋友十分有用，通过这个新的分区功能，DBA 可以定义分区主键增长的步长（间隔），Oracle 会在需要时自动创建新的分区。

- 复合分区种类的扩充。在 11*g* 版本之前，复合分区只支持 Range-Hash、Range-List 这两种方式。而在 11*g* 版本中，复合分区还支持 Range-Range、List-Range、List-Hash 和 List-List 方式，可以说，我们能想到的复合分区种类，基本上都支持了。
- 虚拟列分区。虚拟列的引入使分区更加接近业务，由于虚拟列可以通过实际列的运算得到，因此我们可以设计更加贴近业务的分区模式。
- 引用分区（reference partitioning）。顾名思义，这种分区表和主外键关联有关。比如，order 表和 order_item 表是主从关系的，order 表按照 order_date 分为 4 个区，那么 order_item 表就可以根据外键定义为引用分区，这样从表就会根据主表的分区方式来进行分区。

```
PARTITION BY REFERENCE (order\_items\_fk)
```

- 支持系统管理索引的列表分区（System-Managed Indexes for List Partitioning）。从 11.2 版本开始，列表分区支持系统管理的域索引（domain index）。

对于分区表的使用，网络上存在很多不同的声音，褒贬不一。实际上，任何一种技术都是有其使用价值的，在不同的场合，可能会有不同的效果。这些年来，分区表的优点得到了广大用户的认可，但是到底多大的表才需要分区？对于这个问题，网络上也存在很多观点，但老白对这些观点都不太赞同。我们应该根据具体情况，而不是表的大小，来决定是否使用分区表。在某些场合，即使对一张只有几千字节的表进行分区也是有意义的，比如，一张表的并发更新很频繁，虽然这张表并不大，但是表上所有数据块的 buffer busy waits 等待却十分严重。在这种情况下，如果我们将它设计为 HASH 分区表，那么对于缓解 BBW 等待是很有帮助的。

不同类型的分区表有其不同的应用场合。在某些场合，使用分区表是为了便于维护，将表划分为多个分区，使索引重建和碎片整理更加便捷；在另一些场合，使用分区表是为了在全表扫描时进行分区裁剪，此时需要严格按照 WHERE 语句的裁剪条件来进行分区；而在其他一些场合，使用分区表则是为了便于历史数据归档，这种分区表一般会选择某个时间字段作为分区主键。

本节的最后，老白来介绍一张只有几百条记录的分区表。这是某运营商系统中的一张表，它占用了两三个数据块，包含了仅有的几个统计字段，但大量的并发进程（数百个）会实时更新这张表上的一些统计信息。每当系统业务高峰时，TOP 5 EVENTS 内排在前两位的等待事件中都会出现热块冲突。为了解决这个问题，老白将这张表设计为分区表，并将这 100 多条记录打散到 128 个数据块中，这一极端的做法获得了良好的效果，热块冲突的问题解决了。

12.2　分区表对海量数据的意义

顾名思义，海量数据最大的特点就是数据量多，另外，其扫描开销和数据维护难度也很大。对于海量数据的管理，我们主要会从数据归档、数据扫描效率、数据并发装载性能等方面去考虑。

在不同的应用场合中，设计分区表的类别、选择分区主键、定义分区粒度都是十分关键的。如果我们不能很好地掌握这些技术，那么即使使用了分区表，也可能无法达到预想的目的。本节将从海量数据系统的特点入手，介绍分区表技术的应用场合，以及如何根据应用特点进行适当的设计。

12.2.1　分区表和历史数据归档

历史数据归档是任何一个成熟的应用系统都必须具备的。在一二十年前，软件都带有数据备份和数据归档功能，因为那时大家对数据库技术的研究还不透彻，所以这些功能都必须设计在软件中。另外，由于那时候的磁盘很小，系统的存储容量和 CPU 处理能力都很有限，因此如果数据不及时归档，就很容易出现性能瓶颈。随着数据库和计算机软、硬件技术的高速发展，以及计算机硬件价格的大幅下降，各种 IT 辅助系统日渐完善，如今大多数系统都可以采用自带的或者第三方的产品进行备份和容灾，因此开发厂商对这方面的认识逐渐淡薄，数据归档也逐渐成为可有可无的功能。但随着系统上线时间的推移，数据量越来越大，系统中的碎片也越来越多，系统开销就会越来越大，这也是我们的系统经常随着上线时间的推移而变得越来越慢的主要原因。因此数据归档对于任何一套系统来说，都是十分重要的，从老白多年的项目优化经验来看，至少有60%的系统可以通过数据归档来解决问题。

既然数据归档有这么好的效果，我们就应该经常对自己的系统进行数据归档。但接着下一个问题就来了，在有数据需要归档时，是不是需要采用分区表呢？答案是十分肯定的，分区表很适合需要定期进行数据归档的系统。对于需要历史数据归档的表，第一层分区最好能够按照时间字段分区。

下面来看一个案例。有一张名为 ACC_ITEM_CUR 的表，它是按照 ACC_DATE 字段分区的，每个月我们都会对其历史数据进行归档，归档的方式是从生产表上将数据导入到历史表 ACC_ITEM_HIS 中，然后在生产表上删除这些数据。然而，如果采用传统的方法，我们需要首先将数据插入 ACC_ITEM_HIS，然后用 DELETE 语句删除相关的数据。如果每个月归档的数据是数千万，甚至上亿，那么这项归档工作的开销就是巨大的。

如果比较幸运，ACC_ITEM_CUR 是一张按照 ACC_DATE 分区的分区表，那么我们就可以采用一种更加快捷且开销较小的解决方案了，这个方案就是分区交换（exchange partition）。分区交换是随着分区表在 Oracle 8.0 中出现的技术，其基本原理十分简单，就是通过修改数据字典中的定义，将两个段进行交换。说得具体一点就是先创建一个空段（一张表），然后将分区表的段和这张表的段进行交换，在这个过程中，只需要修改相关段的 DBA_OBJECT_ID，而不需要进行实际的数据交换。通过分区交换，需要归档的分区的段被交换到表上，而 ACC_ITEM_CUR 的这个分区就变为空分区了。下一步，我们再做一次分区交换，将表上包含数据的段交换到 ACC_ITEM_HIS 表的某个分区上，然后再删除 ACC_ITEM_CUR 上的空分区，这样归档操作就完成了，最后进行分区表的全局索引重建操作。

随着 Oracle 版本的提升，分区交换技术也进一步成熟，在 10*g* 和 11*g* 版本中，交换分区支持的分区类型进一步增加，对本地索引交换的支持也日益完善。不过在使用分区交换技术时，也会面临全局索引的问题。由于全局索引可能失效，所以在采用这种技术进行归档的表上，应尽可能减少全局索引的使用，甚至在一些超大的表上，应避免使用全局索引，最好将包括主键在内的索引都设计为分区索引。下面我们通过代码清单 12-4 的示例来回顾一下本节介绍的技术。

代码清单 12-4

```
drop table acc_item_cur;
drop table acc_item_ex;
drop table acc_item_his;

alter session set nls_date_format='yyyy-mm-dd hh24:mi:ss';
--创建生产表
CREATE TABLE acc_item_cur
( acc_item number,
  acc_item_desc varchar2(20),
  acc_date date,
  area_code varchar2(6)
  )
PARTITION BY RANGE (acc_date)
SUBPARTITION BY LIST (area_code)
(PARTITION d1 VALUES LESS THAN ('2012-02-1 00:00:00')
 (SUBPARTITION d1_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d1_con2 VALUES ('AMER'),
  SUBPARTITION d1_con3 VALUES ('AFRICA'),
  SUBPARTITION d1_con4 VALUES ('EUROPE')),
PARTITION d2 VALUES LESS THAN ('2012-03-1 00:00:00')
 (SUBPARTITION d2_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d2_con2 VALUES ('AMER'),
  SUBPARTITION d2_con3 VALUES ('AFRICA'),
  SUBPARTITION d2_con4 VALUES ('EUROPE')),
PARTITION d3 VALUES LESS THAN ('2012-04-1 00:00:00')
 (SUBPARTITION d3_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d3_con2 VALUES ('AMER'),
  SUBPARTITION d3_con3 VALUES ('AFRICA'),
  SUBPARTITION d3_con4 VALUES ('EUROPE'))
);
--添加本地索引，并将其设置为主键
create unique index pk_acc_item_cur on acc_item_cur(acc_item,acc_date,area_code)
local;
alter table acc_item_cur add primary key (acc_item,acc_date,area_code)  using index
pk_acc_item_cur;

--生成一些测试数据，这里老白用了最简单的方法，虽然多了几行代码，但是简单易用
--老白坚持的原则是最简单的就是最好的
create sequence seq_acc_item cache 10000 noorder;
declare
  vdate date;
begin
  vdate:=to_date('2012-01-01','yyyy-mm-dd');
  for i in 1 .. 4000 loop
    insert into acc_item_cur values(seq_acc_item.nextval,to_char(i),vdate,'ASIA');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur values(seq_acc_item.nextval,to_char(i),vdate,'AMER');
  end loop;
  commit;
```

```
  for i in 1 .. 5000 loop
    insert into acc_item_cur
values(seq_acc_item.nextval,to_char(i),vdate,'AFRICA');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur
values(seq_acc_item.nextval,to_char(i),vdate,'EUROPE');
  end loop;
  commit;

end;
/

declare
  vdate date;
begin
  vdate:=to_date('2012-02-05','yyyy-mm-dd');
  for i in 1 .. 4000 loop
    insert into acc_item_cur values(seq_acc_item.nextval,to_char(i),vdate,'ASIA');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur values(seq_acc_item.nextval,to_char(i),vdate,'AMER');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur
values(seq_acc_item.nextval,to_char(i),vdate,'AFRICA');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur
values(seq_acc_item.nextval,to_char(i),vdate,'EUROPE');
  end loop;
  commit;

end;
/

declare
  vdate date;
begin
  vdate:=to_date('2012-03-05','yyyy-mm-dd');
  for i in 1 .. 4000 loop
    insert into acc_item_cur values(seq_acc_item.nextval,to_char(i),vdate,'ASIA');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur values(seq_acc_item.nextval,to_char(i),vdate,'AMER');
  end loop;
  commit;
  for i in 1 .. 5000 loop
```

```
    insert into acc_item_cur
values(seq_acc_item.nextval,to_char(i),vdate,'AFRICA');
  end loop;
  commit;
  for i in 1 .. 5000 loop
    insert into acc_item_cur
values(seq_acc_item.nextval,to_char(i),vdate,'EUROPE');
  end loop;
  commit;

end;
/
--创建中间的交换表
CREATE TABLE acc_item_ex
( acc_item number,
  acc_item_desc varchar2(20),
  acc_date date,
  area_code varchar2(6)
  )
PARTITION BY LIST (area_code)
(PARTITION d1_con1 VALUES ('ASIA', 'AUST'),
 PARTITION d1_con2 VALUES ('AMER'),
 PARTITION d1_con3 VALUES ('AFRICA'),
 PARTITION d1_con4 VALUES ('EUROPE'));
create unique index pk_acc_item_ex on acc_item_ex(acc_item,acc_date,area_code) local;
alter table acc_item_ex add primary key (acc_item,acc_date,area_code)  using index
pk_acc_item_ex;
--创建历史表
CREATE TABLE acc_item_his
( acc_item number,
  acc_item_desc varchar2(20),
  acc_date date,
  area_code varchar2(6)
  )
PARTITION BY RANGE (acc_date)
SUBPARTITION BY LIST (area_code)
(PARTITION d1 VALUES LESS THAN ('2012-02-1 00:00:00')
 (SUBPARTITION d1_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d1_con2 VALUES ('AMER'),
  SUBPARTITION d1_con3 VALUES ('AFRICA'),
  SUBPARTITION d1_con4 VALUES ('EUROPE')),
PARTITION d2 VALUES LESS THAN ('2012-03-1 00:00:00')
 (SUBPARTITION d2_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d2_con2 VALUES ('AMER'),
  SUBPARTITION d2_con3 VALUES ('AFRICA'),
  SUBPARTITION d2_con4 VALUES ('EUROPE')),
PARTITION d3 VALUES LESS THAN ('2012-04-1 00:00:00')
 (SUBPARTITION d3_con1 VALUES ('ASIA', 'AUST'),
  SUBPARTITION d3_con2 VALUES ('AMER'),
  SUBPARTITION d3_con3 VALUES ('AFRICA'),
  SUBPARTITION d3_con4 VALUES ('EUROPE'))
);
```

```
create unique index pk_acc_item_his on acc_item_his(acc_item,acc_date,area_code)
local;
alter table acc_item_his add primary key (acc_item,acc_date,area_code)  using index
pk_acc_item_his;

--进行表分区交换
alter table acc_item_cur exchange partition d1 with table acc_item_ex  INCLUDING
INDEXES;

alter table acc_item_his exchange partition d1 with table acc_item_ex  INCLUDING
INDEXES  WITHOUT VALIDATION ;
```

大家可以看到，最后老白用了两次 EXCHANGE 完成了表分区的交换。在进行分区交换时，老白用到了 INCLUDING INDEXES 子句，这个子句的意思是包含所有索引段的交换。这样做的好处是不需要重建本地索引。这对于数据量较大的场合是十分有用的。

大家可能也注意到了，第二次交换的语法有所不同，多了 WITHOUT VALIDATION 子句。如果我们能够确定数据都是正常的，不需要进行校验，那么就可以使用这个子句。否则，Oracle 会在交换前对 ACC_ITEM_EX 表中的数据做一次全表扫描，以确认不存在不符合分区规则的行。假设每个分区的数据都是亿万级的，那么使用这个子句将十分有效。

12.2.2　分区表和高水位推进

有时，系统可能会出现 enq: HW 或者 enq: FB 这样的等待，这种锁等待是由高水位推进或者格式化数据块引起的。在 RAC 环境下，我们可能还会看到大量的 gc current grant 相关的等待事件，这种等待事件一般也和高水位推进有关。

我们知道，一张表被创建后，在第一次插入数据的时候，会格式化一组数据块，数量一般是 5 个，在早期的 FREELISTS 管理的段空间上，格式化的数据块数量可以通过参数 _bump_highwater_mark_count 来调整，但这个参数在 ASSM 的段中是没有作用的。因为现在已经很少用到 MSSM 了，所以我们主要讨论 ASSM 下这个问题的解决方案。

如果某会话要插入一条记录，而表中格式化过的数据块中已经没有空闲空间可以容纳这条记录了，这时 Oracle 就需要新格式化一组数据块，格式化数据块的数量默认为 5。在格式化完成之前，所有要插入数据的会话都会等待 HW/FB 锁（在 RAC 环境下，会出现 gc current grant 方面的等待）。

要解决高水位推进的问题，最好的方法当然是减少并发插入，将单条数据插入改为集中批处理插入。但是这种修改需要从应用架构方面去做调整，系统上线后较难实施。如果这条路走不通，那么我们就应该反向思考，既然无法减少数据插入的并发量，那么可以考虑通过增加高水位线的数量来解决这个问题，这就和分区表有关了。在前几节大家也了解到了，分区表中的每个分区实际上是一个独立的段，有自己的段头和 HWM，因此如果能够将插入操作均匀分布到多个分区中，那么 HW 的争用就会得到有效的缓解。基于这种考虑，我们需要使用 HASH 分区，因为 HASH 分区能够有效地将序列连续产生的主键映射到不同的分区中。在这种情况下，如果选择范围分区，就无法起到打散数据的作用了。

12.2.3　分区表和 RAC 环境

在 RAC 环境下，分区表的作用十分特别。合理的分区表可以大幅降低实例之间的数据争用，减少 RAC INTERCONNECT 流量，从而达到优化 RAC 系统整体性能的作用。

假设我们有一个应用，是全省用户使用的。在 RAC 环境下，最简单的优化方法是每个地市的用户连接到自己独立的应用服务器组上，而每个地市的服务器组的连接池固定连接到一个实例，这样如果某些表是按照地市编码分区的，而应用软件的 SQL 中又都带有地市编码的条件，那么在一般情况下，某个分区的数据就只会被一个地市的用户访问，而这些用户大多数情况下也只会访问自己本地网的数据。这种分布方式可以有效减少多个实例对同一数据块的争用，从而避免缓冲区融合带来的性能问题。目前电信、移动等运营商都采用这种应用分区的方式。

如果不巧，我们无法按照上面的方法进行分区，而目前某些表上 global buffer busy 等待又十分严重，这时应该如何来优化呢？当然，加大 PCTFREE，减少 BLOCK SIZE 等都是可用的优化方法，但是这些方法的效果都十分有限。如果对这些数据的访问主要以某个字段的=条件为主，很少有<、>之类的查询条件存在（即使存在，执行频率也不是很高，对系统影响相对较小），而这个字段又正好是非空字段，那么我们就可以考虑根据该字段对这张表进行 HASH 分区，这样就可以将数据打散到不同的分区中，从而有效地降低热块冲突。

HASH 分区并不是万能的，它会增加分区字段范围扫描的开销，降低相关访问的效率。因此在设计分区表时，要综合考虑，权衡各种利弊，选择最适合系统的解决方案。

在 RAC 环境中，通过表分区来减少全局热块冲突是十分常用的手段，比如，HASH 分区可以打散数据，减少热块冲突。另外一种常用的分区技术是表分区配合应用分区，从而减少全局消息的数量。比如，在一个全省集中的系统中，每个地市的用户主要访问本地区的数据，极少访问其他地市或者全省的数据。在 RAC 环境中，如果应用分区采用了一个地市的用户固定访问一个实例这样的策略，并且表分区按照地市编码范围或者 LIST 分区，那么就可以限定某个实例的客户端只会访问某些特定分区，这样就可以大大降低全局热块，减少 gcs/ges 的流量。

在解决 RAC 中的全局热块冲突问题时，选择分区的类别十分重要。上例中，我们可以使用 LIST 或者范围分区，当然也可以使用 HASH 分区，但后者的效果就不如前者那么明显了。但在某些情况下，可能 HASH 分区更为有效。比如，应用采用了负载均衡模式，我们无法将访问完全固定到某个实例，这种情况下，彻底打散数据是一个比较简单的实现方案，通过 HASH 分区就可以完成这个目标。

11*g* 版本中新增了一类分区，即系统管理分区。这种分区表没有分区主键，需要程序显式指定分区，将数据插入到某个分区中。实际上，这种分区类似于 HASH 分区，不同的是，应用开发者可以采用自定义的方式将数据分散插入到某个分区中。比如有一张日志表用于记录系统中的一些操作，数据插入后很少被查询，而查询也往往依赖于某些选择性较强的查询关键字进行。这种情况下，就可以将该日志表设计为系统管理分区表。不同的数据库实例插入数据到这张表的不同分区，从而减少 RAC 实例间的冲突访问。

12.2.4　分区主键和分区粒度的选择

在 RAC 环境下，合理使用分区表可以改善系统整体的性能，但不合理的分区往往会导致更为严重的性能问题。分区主键和分区粒度的选择是具有一定技巧的，如果设计不合理，分区表就无法达到预期的效果。

在选择分区主键时，要考虑分区的目的。如果首要目的是数据归档，那么第一层分区肯定要按照时间字段进行范围分区，但如果还有其他的分区目的，可能就需要设计子分区，通过子分区来实现第二目标。无论采用哪种方式设计分区，分区主键是否是系统查询语句中经常使用的过滤条件，这一点十分重要。比如，在一张表中，PHONE_NO 是一个全局性的号码，只能属于某个本地网（地市），而系统中有一张根据本地网编码分区的分区表。对于这张表，我们可以只通过 PHONE_NO=...来访问，因为 PHONE_NO 具有全局意义，一个 PHONE_NO 肯定只能属于一个本地网。这种查询在语义上没有任何问题，但是由于过滤条件中没有本地网代码，所以无法使用分区裁剪等功能。如果在 WHERE 条件中加上 AND LAN_ID = ...，那么这条 SQL 就能够用到分区裁剪功能了。

基于上面的描述，大家可能已经对分区主键的选择有所了解了，就是要根据访问这张表的 SQL 中的 WHERE 条件来进行设计。这对于开发厂商来说相对比较容易，而对于那些对软件细节一无所知的 DBA 来说，可能就很难了。我们应该如何在黑盒中进行分析呢？其实这个工作难度并不大，只是比较费时。通过分析共享池中的 SQL，查看过滤条件和连接条件，就可以判断出哪个分区主键是比较合适的。代码清单 12-5 是老白常用的脚本（这个脚本是 Steven Adams 编写的，早期学习 Oracle 的朋友应该比较熟悉 Steven，他的 ixora 网站曾经是大多数想学习 *Oracle Internal* 的 DBA 必须光顾的，老白对这个脚本进行了一些修改）。

代码清单 12-5

```
accept OwnerName prompt "Owner Name: "
accept TableName prompt "Table Name: "

column sql_text format a58 word_wrapped

select /*+ ordered use_hash(d) use_hash(c) */
  kglnahsh     hash_value,
  sum(c.kglobt13)  disk_reads,
  sum(c.kglobt14)  logical_reads,
  sum(c.kglhdexc)  executions,
  c.kglnaobj  sql_text
from
  sys.x$kglob  o,
  sys.x$kgldp  d,
  sys.x$kglcursor  c
where
  o.inst_id = userenv('Instance') and
  d.inst_id = userenv('Instance') and
  c.inst_id = userenv('Instance') and
  o.kglnaown = upper('&OwnerName') and
```

```
    o.kglnaobj = upper('&TableName') and
    d.kglrfhdl = o.kglhdadr and
    c.kglhdadr = d.kglhdadr
  group by
    c.kglnaobj,kglnahsh
  order by   3
  /
```

通过这个脚本，可以将访问某张表的所有 SQL 都查找出来，并根据逻辑读的数量进行排序。（当然你也可以修改 order by 语句，选择其他的排序字段。比如，order by 2 是按照物理读排序，order by 4 是按照执行次数排序，这些都是最为常用的排序方法。老白一般会根据执行次数和逻辑读进行排序分析。）从后往前分析，如果发现绝大多数查询的过滤条件中都包含某个字段，该字段就可以作为分区关键字的候选对象。

有了候选对象，下面就好办了。如果我们能够确定该字段非空，那么就可以进一步确定其作为分区主键的可行性。如果能够和开发团队沟通，进一步从应用系统本质的角度来验证，那就更好了。明确了分区主键，下一步就要明确分区的粒度。一个表分区多大才算合适呢？有人说 500 万条记录，也有人说 1000 万条记录，还有人说 100 万条以内。实际上，简单地用记录数来确定表分区的粒度过于轻率。表分区是为某个目的服务的，因此如何更好地服务于目的才是选择粒度的最根本要素。

如果分区的目的是为了便于维护，那么我们就需要明确，多大的分区维护起来才比较便捷。如果可用于维护的时间窗口很短，所有的分区维护操作要求在 30 分钟内完成，那么可能 500 万条记录以下是比较好的选择，否则我们很难在一个维护窗口内完成所有的操作。

如果分区的目的是为了便于归档，那么归档周期就是分区粒度设计的一个很重要的因素。分区的跨度不能超过一个归档周期，不过一个归档周期内可以包含多个分区。再结合其他因素，我们就能很容易地找到合适的分区粒度。

如果分区的目的是为了提高性能，那么我们就要考虑这张表的范围扫描大多发生在怎样的时间范围内，尽可能将分区粒度和这个扫描范围进行对应。10 年前老白曾碰到过一个案例，当时客户找了很多人都分析不出问题的原因。这个系统的大多数表是按照季度分区的，在每个季度初的 5 号～10 号，系统的 I/O 负载总是特别高，系统性能也非常差，但过了 10 号系统就开始转好，15 号后就恢复正常了。老白分析后发现，这个系统的范围扫描一般集中在最近半个月内，在每个季度最初的几天里，总是要在本季度和上季度的分区中扫描数据，这样，一些做分区扫描的应用的开销就会比平时大（要扫描两个分区），而 15 号后，扫描上一个分区的应用减少，性能就恢复了。于是，老白建议对分区的粒度进行调整，修改为每半个月一个分区。这样最坏的情况也只是需要对两个半月的数据进行扫描，与以前扫描 3 个半月的数据相比，改善了很多。确实，分区调整完毕后，每季度一次的性能危机就没有再发生了。

第 13 章

理解序列

13

序列（sequence）是 Oracle 的序列号发生器。很多表中必须存在唯一性的主键，主要用于业务关联查询和防止重复。大多数主键都是单字段的，而且没有具体的含义，只是用来相互区分、保证唯一性而已，这种字段的取值就很适合使用序列。尽管序列被广泛使用，但由于其使用方法十分简单，很多人对它都不太在意。实际上，在很多场合下，序列的性能也会对系统产生很大的影响。在我们设计应用时，合理地设计序列也是十分必要的。

13.1　什么是序列

在使用序列之前，老白都是自己设计序列号发生器的，创建一张表，然后在表中维护一组记录，一个典型的设计如表 13-1 所示。

表　13-1

字　　段	类　　型	说　　明
Id	Integer	序列号发生器的编号，唯一的主键
Val	Integer	当前值
Desc	Varchar2(20)	说明
Module	Varchar2(20)	使用这个发生器的模块

在这种结构中，我们通过 id 来访问对应的序列号。一般来说，会设计一个函数来获取序列号，首先通过 select ... From ... For update 来锁定这条记录，然后将序列号的 Val 增加 1。实际上，Oracle 的序列也是通过类似的机制来实现的。大家可以通过 10046 trace 很方便地看到序列的创建过程。这个过程十分简单，下面老白将通过一个示例来展示。

比如，在 SYSTEM 用户下执行：

```
Create sequence seq_test cache 100 noorder;
```

首先检查该用户下的 Object 是否存在重名：

```
select obj#,type#,ctime,mtime,stime,status,dataobj#,flags,oid$, spare1, spare2 from
obj$ where owner#=:1 and name=:2 and namespace=:3 and remoteowner is null and linkname
is null and subname is null
```

如果没有重名的，系统就会继续，并通过 obj$中的隐含行（_NEXT_OJBECT)获取到 obj#，然后进行操作：

```
select increment$,minvalue,maxvalue,cycle#,order$,cache,highwater,audit$,flags from
seq$ where obj#=:1
```

通过上面的 SQL，可以确认该 obj#在 seq$中不存在，于是就插入一条新的记录：

```
insert into seq$ ( obj#, increment$, minvalue, maxvalue,cycle#,
order$,cache,highwater,audit$,flags)values(:1,:2,:3,:4,:5,:6,:7,:8,:9,:10)
```

当然，生成了 seq$的数据后，还要创建 obj$的数据，否则就会出现数据字典不一致：

```
insert into obj$ ( owner#, name, namespace, obj#, type#, ctime, mtime, stime, status,
remoteowner, linkname, subname, dataobj#, flags, oid$, spare1, spare2) values
(:1,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14,:15,:16,:17)
```

在使用序列时，第一次访问会将序列的信息从 seq$中查出，存放在内存中：

```
PARSING IN CURSOR #1 len=102 dep=1 uid=0 oct=3 lid=0 tim=1303870556391017 hv=3967354608
ad='68acc64c'
select increment$,minvalue,maxvalue,cycle#,order$,cache,highwater,audit$,flags from
seq$ where obj#=:1
END OF STMT
PARSE #1:c=0,e=22,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=4,tim=1303870556391012
BINDS #1:
kkscoacd
 Bind#0
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
  kxsbbbfp=b72a4388  bln=22  avl=04  flg=05
  value=72384
```

第一次访问序列是比较特殊的，会马上修改 seq$中的数据，而随后，序列号是在共享池中进行分配的，不需要每次都去访问 seq$了，只有 CACHE 耗尽，才会再次 update seq$，将 highwater 的值加大：

```
PARSING IN CURSOR #1 len=129 dep=1 uid=0 oct=6 lid=0 tim=1303870556392458 hv=2635489469
ad='689bdee4'
update seq$ set increment$ =:2, minvalue=:3, maxvalue=:4, cycle#=:5, order$=:6,
cache=:7 ,highwater=:8, audit$=:9,flags=:10 where obj#=:1
END OF STMT
```

由于存在 CACHE，序列机制比老白早期自己维护的序列号发生器的效率要高很多。实际上，老白后来也优化了序列号发生器，采用了类似序列的机制，在共享内存中维护一个缓冲区，并发访问的进程通过信号灯来进行同步。序列确实很简单，只要是合格的架构师，就能设计出类似的机制。

13.2 序列的使用和优化

序列号发生器包含 seq$基表和一个缓冲机制，每次访问 seq$基表都会分配一组缓冲，缓冲用

完之前不会再去访问 seq$基表。这一点大家可以通过 10046 TRACE 来验证。正因如此，序列才会在数据库实例重启后产生跳号现象。如果我们的应用不允许跳号产生，那么就只能通过将序列设置为 NOCACHE 来解决了。根据这一点，大家可能很快就会想到，在实际应用中，如果我们为序列设置较大的 CACHE，就能减少访问 seq$基表的次数，从而提高序列的访问效率。表 13-2 是老白在一个项目中进行的测试。

表　13-2

CACHE	并发数量	执行次数	测试时间（毫秒）	平均每次执行时间
2	8	200001	942122	4.7
100	8	200001	73282	0.366
1000	8	200001	48250	0.241
5000	8	200001	47129	0.236
20000	8	200001	43299	0.216

从结果来看，CACHE 从 2 提高到 100，序列的访问性能提升了 10 多倍；从 100 提升到 1000，性能提升了近一半；而从 1000 到 20000，性能提升的幅度就很有限了。从实验数据可以看出，设置较大的 CACHE 能够有效地提升序列访问性能。

除了 CACHE 这个属性外，序列还包含一个和性能关系极大的属性 ORDER，其含义是每次通过.nextval 获取序列的下一个值时是否排序。如果我们设定某个序列是有序的，那么每次申请 nextval 时，根据提出申请的时间，产生的序列值是严格排序的，而无序则没有这个限制。ORDER 属性要求序列中存在一种完全串行的机制，每次只能分配一个序列号，否则就无法实现严格排序。ORDER 属性是和业务相关的，如果业务要求序列号是严格排序的，那么就必须使用 ORDER 属性；但如果序列号只是用作主键，顺序是无所谓的，那么 NOORDER 就是最好的选择。

在 RAC 环境中，ORDER 属性会带来很严重的性能问题，因为 Oracle 要保证整个 RAC 范围内序列号的产生都是按顺序的。RDBMS 必须在 RAC 环境中建立一种串行机制，来协调序列号的产生，这种需求会大大降低序列号 CACHE 的作用。因为序列号发生器每次申请一组连续的号码，另外一个实例的序列号发生器也会申请一组连续的号码，这两组号码不会重复交叉，并且是有先后顺序的。序列号只能按组使用，开始时，所有的序列号申请都从较小一组的 CACHE 中进行分配，这个 CACHE 分配完了，再申请下一组，而这一组的最小值大于另外一个实例缓冲中的最大值。因此接下来的所有申请都从另外一个实例中分配，如此循环。

从这个角度来看，使用 ORDER 属性的代价是很大的，因此在一般的情况下，不建议设置 ORDER 属性，特别是在 RAC 环境下。幸运的是，NOORDER 是默认的设置，因此在绝大多数开发人员和 DBA 不知情的情况下，系统已经使用了性能较好的选项。

总的来说，序列号是十分简单的，只要设置较大的 CACHE（在一些大型的系统中，可能需要设置 1000，甚至 5000），并尽可能使用 NOORDER 选项，就不会出现太大的问题。不过也有一些系统，即使设置了很大的 CACHE，也使用了 NOORDER 选项，还是会存在十分严重的序列冲突。这种情况下，我们可以考虑为应用设置多个序列，这样，应用就可以根据某种分区规则，

访问不同的序列号发生器来产生序列号，然后再在主键中拼入不同的分区标识，来实现主键的唯一性。比如，将应用分为三个区 A、B、C，通过 A、B、C 这三个字母和序列号拼接形成完整的主键，如 A0012345、B0023456 等。相关示例如下所示：

```
SELECT 'A'||TO_CHAR(SEQ1.NEXTVAL,'099999999') FROM DUAL;
SELECT 'B'||TO_CHAR(SEQ2.NEXTVAL,'099999999') FROM DUAL;
SELECT 'C'||TO_CHAR(SEQ3.NEXTVAL,'099999999') FROM DUAL;
```

Part 2 第二部分

分析思路篇

思路是分析问题过程中最为关键的要素，如果没有思路，那么我们就只能漫无目的地猜想，无法将焦点集中在关键的地方。这会导致问题定位时间长，解决效率低下，甚至出现治标不治本的现象。

很多 DBA 都有这样的感觉，开始的时候好像一直在黑暗中摸索，突然的灵感或者某个人的指点，就会让他茅塞顿开，很快找到问题的关键，从而打开解决问题之门。

我们如何捕获这转瞬即逝的灵感呢？不断地学习当然是很重要的，但更为关键的是理解 Oracle 的基本原理，每次分析问题时都能够抓住重点，尽快找到解决问题的最佳途径。

但我们不可能每次都很幸运地抓住问题的要点，这种情况下，建立自己的预案体系就十分重要了。后续章节中的一些内容可以作为 DBA 建立分析预案体系的参考。

第 14 章 问题分析综述

DBA 往往面对的是一个十分复杂的系统，涉及面非常广泛，相关技术细节也比较复杂。如何在纷杂的情况下尽可能避免风险，并快速解决问题呢？本章我们将从以下几个方面来讨论这个问题。

第一，当面对看上去十分复杂的情况时，应该把握好问题的本质。从纷繁的表象中发现问题的本质是十分困难的事情，这和 DBA 的能力与知识结构有很大的关系。另外一方面，DBA 的工作态度也对问题处理结果影响很大。14.1 节“如何抓住蝴蝶效应中的那只蝴蝶”通过一个案例阐述了如何在纷繁的表象下凭借细致的工作，最终找到问题的本源。要想抓住那只扇动翅膀的蝴蝶，需要的不仅仅是技术，更重要的是责任心。因此，分析问题时，只有责任心强的 DBA 才会打破砂锅问到底，最终发现问题的本质。

第二，如果对 Oracle 的本质缺乏了解，那么即使再有责任心，也无法抓住问题的本质，因为你根本不知道某个表象后面隐含的 Oracle 基本概念。因此为了能够提高分析和处理问题的能力，我们必须加强自己对 Oracle 基础概念的认知，只有了解了 Oracle 的基本原理，才能更好地分析问题。

第三，经验的积累。如果一个问题是你以前见过的，或是能在你的知识库或者 Metalink 上找到的，那么这个问题的分析过程会大大缩短，投入的成本也会小得多。很多 DBA 都不注意日常的知识积累，总是把发生过很多次的问题当作新问题来处理，只有分析到后期才突然发现这个问题以前好像碰到过，但是在查找以往的文档时，又找不到相关的信息，于是只好重新再处理一遍，这样就大大增加了工作量。其实，积累知识库是 DBA 日常工作中十分重要的一部分，如果每次处理问题后都能够认真总结，并且梳理处理的过程，检讨可以改进的地方，那么时间长了，就能够积累大量的案例和处理预案，从而提高处理问题的能力。在这方面，老白的建议是写博客，这样既能够及时总结经验，又能够锻炼思维的周密性，还能够学习一些之前自己模棱两可的知识（在网络上发布文章，总不能错误连篇，令人汗颜吧？），时间长了，也许还能成就一个网络名人。

第四，分析问题的成本。对于任何一个问题，如果不计成本地去分析，那么找到问题根本原因的概率还是很高的，但并不是所有的问题都值得这样做。只有那些已经或者可能对系统产生较大影响的问题才值得投入较大的成本去分析。有限的资金要用到最关键的地方，否则 DBA 就会陷入一个又一个谜团之中，无法正常管理和维护系统了。

14.1 如何抓住蝴蝶效应中的那只蝴蝶

南美丛林中的一只蝴蝶扇动翅膀，可能导致莫斯科下大雪，这说明了大气系统的复杂性。而DBA 在日常工作中也经常会面临类似的问题，我们只从故障的表象上分析和处理问题，采取的措施仅仅针对一些表象，就没有找到问题的关键。也就是说，我们并没有抓到扇翅膀的那只蝴蝶，而仅仅抓住了莫斯科上空的乌云。

老白在写这本书时碰到了一个案例，写出来和大家分享。客户的一套系统下午 1 点多时，突然出现了故障，服务无法响应，新会话连不上去。最后杀掉了大量的会话，才恢复正常。客户很想知道问题的原因，找到我时已经是下午 4 点多了。出现故障的时段有大量如下所示的信息：

```
Mon Apr 11 12:52:24 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_10410.trc:
ORA-00603: ORACLE server session terminated by fatal error
ORA-27544: Failed to map memory region for export
ORA-27300: OS system dependent operation:bind failed with status: 227
ORA-27301: OS failure message: Can't assign requested address
ORA-27302: failure occurred at: sskgxpcre3
Mon Apr 11 12:55:01 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_13426.trc:
ORA-00603: ORACLE server session terminated by fatal error
ORA-27544: Failed to map memory region for export
ORA-27300: OS system dependent operation:bind failed with status: 227
ORA-27301: OS failure message: Can't assign requested address
ORA-27302: failure occurred at: sskgxpcre3
Mon Apr 11 12:55:25 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_13934.trc:
ORA-00603: ORACLE server session terminated by fatal error
ORA-27544: Failed to map memory region for export
ORA-27300: OS system dependent operation:bind failed with status: 227
ORA-27301: OS failure message: Can't assign requested address
ORA-27302: failure occurred at: sskgxpcre3
Mon Apr 11 12:55:25 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_13936.trc:
ORA-00603: ORACLE server session terminated by fatal error
ORA-27504: IPC error creating OSD context
ORA-27300: OS system dependent operation:bind failed with status: 227
ORA-27301: OS failure message: Can't assign requested address
ORA-27302: failure occurred at: sskgxpcre3
Mon Apr 11 12:55:25 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_13938.trc:
ORA-00603: ORACLE server session terminated by fatal error
ORA-27504: IPC error creating OSD context
ORA-27300: OS system dependent operation:bind failed with status: 227
ORA-27301: OS failure message: Can't assign requested address
ORA-27302: failure occurred at: sskgxpcre3
Mon Apr 11 12:56:00 2011
Thread 2 advanced to log sequence 2945 (LGWR switch)
  Current log# 4 seq# 2945 mem# 0: /redolog/test2/redo04.log
Mon Apr 11 12:56:01 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_14554.trc:
```

```
ORA-00603: ORACLE server session terminated by fatal error
ORA-27544: Failed to map memory region for export
ORA-27300: OS system dependent operation:bind failed with status: 227
```

同时还存在一些类似的 ORA-27*XXX* 错误：

```
Mon Apr 11 12:56:33 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_22957.trc:
ORA-27509: IPC error receiving a message
ORA-27300: OS system dependent operation:recvmsg failed with status: 216
ORA-27301: OS failure message: Socket operation on non-socket
ORA-27302: failure occurred at: sskgxprcv1
Mon Apr 11 12:56:33 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_25431.trc:
ORA-27509: IPC error receiving a message
ORA-27300: OS system dependent operation:recvmsg failed with status: 216
ORA-27301: OS failure message: Socket operation on non-socket
ORA-27302: failure occurred at: sskgxprcv1
Mon Apr 11 12:57:24 2011
```

根据 ORA-27300: OS system dependent operation:recvmsg failed with status: 216 的错误描述，现场工程师认为这是 bug 6689903 导致的，建议关闭 NUMA。客户准备晚上实施此操作，想听听我的建议。我觉得关闭 NUMA 是一个很重要的操作，应该十分谨慎，所以先要搞清楚到底是什么导致了今天的问题。从 ORA-27300 来看，这是由于某种 OS 资源不足导致的。因此我们首先要从错误信息入手，分析 HP-UX 的 ERRNO=227、ERRNO=216 这两条错误信息，查阅 HP-UX 的 errno.h 文件：

```
#  define ENOTSOCK              216     /* Socket operation on non-socket */
#  define EADDRNOTAVAIL         227     /* Can't assign requested address */
```

216 错误表示在非 Socket 上进行了 Socket 操作，227 表示无法分配地址。Oracle 官方对 bug 6689903 的解释是，使用了 NUMA 后，Oracle 存在一个 Bug，导致一个会话使用了大量的 UDP 端口，造成 UDP 端口不足，可以通过更新补丁或者关闭 NUMA 来解决这个问题。由于 UDP 端口耗尽也可能导致 ERRNO=227 错误，因此可以初步判定就是 UDP 端口耗尽导致了问题。在这种情况下，更新 PATCH 6689903 只能解决过多 UDP 端口被一个会话消耗的问题，并不一定能解决所有的问题，当系统负载进一步加大时（系统设置的 PROCESSES=4500，而发生故障时会话数无法突破 1600），可能还会出现问题。而关闭 NUMA 虽然可以减少 UDP 端口的使用，但是会降低系统的性能，无法充分发挥大型 SMP 系统的架构优势，也是不足取的。因此较好的解决方案是先更新 PATCH 6689903，避免 Bug 过多消耗 UDP 端口，另外调整 UDP 端口的范围，从而使 OS 提供更多的端口，可通过下列命令完成此操作：

```
oracle@test22:/usr/include/sys$ ndd -get /dev/udp udp_largest_anon_port
65535
oracle@test22:/usr/include/sys$ ndd -get /dev/udp udp_smallest_anon_port
49152
```

我们看到系统的 UDP 端口使用了默认值，通过调整这两个值，使中间的区间变大，OS 就能提供更多的 UDP 端口号了。问题分析到这里，已经解决得差不多了。可能大多数 DBA 会觉得大

功告成了，但老白认为不然，如果说建议关闭 NUMA 只是看到了下雪时莫斯科上空的乌云，那么分析到这里也仅仅是看到了西伯利亚冷空气的影响，离那只南美洲的蝴蝶还有万里之遥呢。

老白当然会继续分析下去，是什么原因导致 UDP 端口号被耗尽了呢？客户说，平时这个系统会话数在 1000 出头，故障时却达到了 1600。这很好地解释了刚才的问题。但是为什么会话数会突增呢？通过对应用架构的了解，我们发现这个系统的大多数应用没有采用连接池，而是客户端直连的，当系统处理能力下降时，客户端与数据库之间的连接会增加，以适应外部服务的请求。因此，我们可以将焦点转移到使系统变慢的情况。如果在故障前的某个时段，系统突然变慢了，那么就有可能造成会话数增加。会话数增加后，UDP 端口配置过低的问题就暴露出来了。

那么接下来就需要分析系统为什么会变慢，是在什么时间变慢的。通过继续分析 ALERT LOG，可以发现第一次报错的时间是 12 点 41 分左右。

```
Mon Apr 11 12:38:06 2011
Thread 2 advanced to log sequence 2940 (LGWR switch)
  Current log# 3 seq# 2940 mem# 0: /redolog/test2/redo03.log
Mon Apr 11 12:40:58 2011
Errors in file /oracle/app/oracle/admin/test2/udump/test22_ora_25451.trc:
ORA-00603: ORACLE server session terminated by fatal error
ORA-27544: Failed to map memory region for export
ORA-27300: OS system dependent operation:bind failed with status: 227
ORA-27301: OS failure message: Can't assign requested address
ORA-27302: failure occurred at: sskgxpcre3
Mon Apr 11 12:40:59 2011
Trace dumping is performing id=[cdmp_20110411124059]
```

看来故障点应该在 12 点 41 分之前。于是我们做了一个 ASH 报告，来查看 12:00 到 12:40 之间系统发生了什么。为了便于分析，先按照 10 分钟周期生成 4 个报告，在前面 3 个报告中，一切正常，而在 12:30 ~ 12:40 的报告中，我们发现了一个疑点：

```
gcs drm freeze in enter server       24
```

在 1 分钟内活跃会话的采样中，出现了 24 次 DRM 等待，平均等待时间为 600 毫秒左右，而且这个时间段内的 SQL 执行次数、BUFFER GET 等指标明显低于前面的时段。因此我们可以初步断定，这可能是导致会话数量突增的一个重要原因。但是这个系统的另外一个节点跑的是完全不同的应用，而且还没有投产，为什么会出现这么多 DRM 呢？通过 LMD、LMON、LMS 等日志，我们发现，12:36 ~ 12:38 这段时间里的 DRM 数量比前面的时段增加了数倍。于是，我们对另外一个节点上的 12:30 ~ 12:40 时间段生成了一个 ASH 报告，在这里终于看到了那只美丽蝴蝶的真面目。原来在这个时段，另外一个节点上有人用 SQLPLUS 登录，访问了大量的故障节点数据。而这个操作导致了 DRM 事件增加，短暂降低了系统的性能。如果 UDP 端口号充足，这种影响不会被放大，而仅仅会在 12 点多钟业务不繁忙时段出现短短几分钟的性能下降，很快就会平息。但正是由于 UDP 端口号不足，才放大了蝴蝶扇动翅膀的动作。

抓住了这只蝴蝶，那么如何解决这个问题就很明显了，尽可能不要出现类似的操作是肯定的。不过另外一个问题也是需要我们考虑的，在这样的一个系统中，DRM 其实是不必要的，因为正常情况下，两个节点会跑自己的数据，不会交叉，因此关闭 DRM 是一种更理想的选择。

大家可能对关闭 DRM 这个结局感到意外，但是如果你看过了抓蝴蝶的全过程，就会认为这是顺理成章的了。

事情就是这么简单，但是我想大多数人只会走到这个过程中的某个步骤。这就是 DBA 之间的差距，不仅仅是技术上的，更多的是态度的问题。

14.2 为什么要强调基础概念

在学习 Oracle 的过程中，我得到一个十分重要的经验，那就是从普通 DBA 成长为专家、高手，其中非常重要的一环就是对基础概念的理解和精确掌握。当然我所说的精确掌握，并不是要求每个 DBA 都把 Oracle 的源代码读一遍，了解其最底层的处理模式和方法。你可以了解得不那么深入，但是必须十分精准地掌握所学的知识。

在 10 多年前，我是靠经验来处理问题的，那时互联网还没有普及，除了 Metalink，没有太多很好的技术资料来源。因此自己在处理问题时往往不得要领，有时候要绕很多弯路才能找到正确的方向。而我现在处理问题，一般都会直奔主题，很快就能抓住问题的关键。这和 10 年前那一次长达两年的基础知识整理和学习是密不可分的，这些经历让我这 10 年都受益匪浅。

前几天我又接到了一个 morning call。做维保的人最怕这种凌晨三四点钟的电话，这种电话一般都意味着故障，而且往往是大故障。电话铃声一响，我就从床上坐了起来，接到电话后，提着的心才放了下来。原来这个客户的监控平台报警，有一个数据库在用户登录时提示某个表空间不足，这是一个应用的表空间，而不是系统表空间。发现这个故障后，客户没有直接通过扩充表空间来解决问题，而是想分析一下为什么会出现这样的故障。可是他研究了很长时间也没有头绪，所以希望我能帮他分析一下问题的原因。

事实上，这个客户处理问题的方法十分正确，可能很多 DBA 碰到这类的问题都会直接扩充不足的表空间了事。这种头痛医头、脚痛医脚的做法，往往会让我们失去很多发现深层次问题的机会，甚至留下更为严重的、灾难性的隐患。

对于这个问题，我考虑了一下，根据数据库登录机制的基本原理，这中间应该很少会涉及用户表空间，除非在登录过程中有客户化定制的处理步骤。因此查找问题的方向一般只有两个，一个是 LOGON 触发器或者其他系统级触发器，另一个是审计。

LOGON 触发器是系统在用户登录时调用的一个可客户化定制的模块，如果客户在 LOGON 触发器中写入了一些类似于日志的信息，那么就有可能出现类似的结果。一般来说，默认的审计表不会存放到应用表空间中，不过有些用户可能会将审计表从系统表空间移到用户表空间。不过从报错信息来看，需要扩展的表和审计无关，因此可以初步排除审计的问题。

基于上述分析，我建议客户检查一下系统中是否存在 LOGON 触发器。用户检查了 SYS 账号后，并没有发现 LOGON 触发器。这让我感到有点困惑，根据我对登录过程的理解，一般来说如果排除了审计，唯一的可能就是系统级触发器。因此，我建议他再仔细检查一下触发器，我马上通过 VPN 连上去看看。1 分钟后，我的 VPN 拨号还没完成，那边的电话又打过来了，问题找到了，在一个拥有 SYSDBA 权限的普通用户上，发现了一个 LOGON 触发器。

这个故障的处理时间大概在 5 分钟到 10 分钟之间，我的头还昏昏沉沉的，问题就解决了，倒在床上，5 分钟后我又进入了睡眠状态。为什么能这么快定位到这个问题呢？这就是基于对相关知识的把握。我不需要详细记住 LOGON 触发器具体都做了哪些操作，而只需要知道登录过程中，哪些步骤可能会产生写操作，哪些地方是可以个性化定制的，这样就完全可以解决问题了。基于这些知识，我们很快就把重点放在了对触发器的检查上，虽然中间也犯了一个小错误，刚开始仅仅检查了 SYS 账号。实际上，这只是我在相关知识的掌握方面存在的一点瑕疵所导致的。在我的印象里，LOGON 触发器都是在 SYS 账号下创建的，很少有人在其他用户下创建 LOGON 触发器。但事实上，只要有相应权限的用户都可以创建系统级触发器，而且由于 LOGON 触发器是系统级的，因此无论在哪个用户下创建的 LOGON 触发器，都会被系统事件激发，因此在查找触发器时不能仅仅局限于 SYS 用户，还要考虑所有拥有 DBA、SYSOPER 等超级权限的用户。如果当时使用下面的 SQL 去查找，可能会更快地找到问题的原因：

```
SQL>COL TRIGGERING_EVENT FORMAT A30 TRUNC
SQL>SET LINE 132
SQL>select owner,trigger_name,trigger_type,triggering_event from dba_triggers

OWNER   TRIGGER_NAME         TRIGGER_TYPE     TRIGGERING_EVENT
------- -------------------- ---------------- ------------------
SYS     SYS_LOGON            AFTER EVENT      LOGON
SYSTEM  LOGON_AUDIT_TRIGGER   AFTER EVENT     LOGON
```

从上面的案例可以看出准确掌握基础知识的重要性，同时我们也看到了，准确掌握并不是深入掌握。实际上，解决大多数问题，都不需要很深入地掌握内部原理，仅仅掌握表层的原理就够用了。Oracle 数据库是一个十分庞大的软件体系，一个问题可能牵涉的技术要点非常多，如果没有重点地进行分析，就和撞大运差不多了，处理问题的效率会很低下，找到问题根源的机会也会很少。掌握原理，可以帮助我们很快地将焦点集中在重点区域，从而大大提高解决问题的效率。

最近这几年我已经很少做实际操作了，做的最多的事情就是接电话，然后提出建议。通过几次电话交流，大多数问题都能被一线的工程师解决。这些工程师的实际动手能力都不错，事情交给他们来完成是没有任何问题的，但是他们在碰到问题时，往往很难快速地找到问题的关键点，在这种情况下，通过一些准确的指导，就可以让他们找到正确的方向，从而很快地解决问题。有时候，我们回顾问题的解决过程时，经常会发现，很多一线的工程师在具体贯彻我提出的思路时，会进行很多创造性的工作，即便我对这个问题原理的理解更准确、深刻，但很多问题如果换成我来做，处理的结果可能并没有他们好。这是因为他们长期在第一线从事第一手的操作工作，在某个方面的具体操作层面上，他们的实际经验可能比我还要丰富。准确把握基础原理和丰富的实际操作经验，是更好地解决问题的两个必要条件，如果这两点结合得好，处理问题就会事半功倍。在完成一项工作时，一个优秀的一线操作人员和一个资深专家的组合往往是最佳的。很多客户都希望所有的事情都能由资深专家亲自操刀，其实这不一定是很好的选择，专家可能更多的是深入研究一些原理性的问题，实际操作能力有可能已经相对下降了，操作的效果可能还不如一线操作人员。

在这种情况下，最好的方法是，专家指导完成操作手册，由一线操作人员实施具体的操作。

14.3　工作中的好习惯带来的福利

在本节开始之前，我先提一个问题：如果客户的 Oracle 安装目录满了，udump 下有好多几百兆字节甚至上千兆字节的 trace 文件，而这个时候，客户的存储上又没有空闲空间备份这些文件了，而且情况十分紧急，已经有业务受到了影响，需要尽快解决问题，这种情况下该怎么处理？

这个问题看似十分简单，不过要想让老白打满分也确实不容易。我想大多数 DBA 会直接清了 udump 目录了事，也有一些相对谨慎的 DBA，会先把这些文件备份，然后再删除 udump 目录，但因为本地没有可用的备份空间，于是他们会通过 ftp 将这些文件先复制到自己的电脑上。还有一些 DBA 处理得更加“艺术”，首先通过 ftp 复制几个不是特别大的 trace 文件，然后删除 udump 目录，让系统故障先消失，再慢慢处理那些比较大的文件。

其实，如果不将现场的紧急情况作为条件，解决这个问题的好方法有很多。但是在很紧急的情况下，要采用最为正确的方法确实很难。老白就碰到过这样的问题，而且差点犯了一个低级的错误，幸亏多年养成的好习惯帮了忙，否则后果不堪设想。

问题场景和我上面的描述十分相似。这是一个流复制的项目，系统刚刚上线，我正在现场值守，由于 SDH 线路问题以及目标端 APPLY 进程很缓慢，所以一直在调整 APPLY 进程的性能。这个时候归档目录突然满了，但由于 CAPTURE 的 REQUIRED_CHECKPOINT_SCN 还没推进到某个点，所以目录下的归档日志已经无法删除了。经过检查，udump 下有几千个 trace 文件，最大的有几千兆字节，加在一起有七八千兆字节，于是我和客户商量了一下，决定马上清掉 udump 目录。由于当时无法归档，业务系统已经挂起了，我很紧张，没怎么考虑就执行了 rm 命令。命令执行完了，用 df 命令一检查，发现文件系统使用率还是 100%。当时脑子“嗡”的一下，汗马上顺着脖子流了下来。我突然想到一个问题，在 UNIX 系统下，如果一个文件被删除了，但某个进程打开这个文件后没有关闭，那么该文件可能就无法释放空间。由于我删除的是 udump 下的文件，所以还必须把相关的前台进程也杀掉。因此我需要找到那几个最大文件的文件名，根据文件名里的进程号去杀掉相关进程。于是我马上把 secureCRT 的屏幕往前翻，去查找刚才 ls －l 命令的结果。

屋漏偏逢连夜雨，因为翻得太急，secureCRT 居然死掉了，拖动屏幕没有任何反映。由于刚才没有备份相关的 trace 文件，所以我根本不知道那几个比较大的 trace 文件的文件名是什么。重新启动 secureCRT 登录系统后，我发现这套系统没有安装 lsof，也没有安装 glance 之类的系统工具。看来我碰到大麻烦了，几千个会话，不知道要杀掉多少才能解决这个问题，如果全部杀掉，那就是大故障了。当时我很后悔没有把 ls 的信息记录下来。这时我突然想到，我的 secureCRT 应该是开了会话日志的，而且是追加模式的。于是我通过 secureCRT 找到了那个日志以及刚才执行 ls –l 的结果。杀掉几个会话后，磁盘空间终于释放了。

10 多年前，有一次我在客户现场安装软件时，突然服务器上有文件被误删了，客户认为是我干的，但我很肯定这与我无关，因为我访问的磁盘都和那个被误删的文件不同。但是我没有任何证据能证明自己的清白，最后的结果是和客户吵了一架，客户向领导投诉了我。从那以后，在用户现场，我一般都会连入客户的服务器，并打开会话跟踪日志，每次安装完 secureCRT 后马上

将其设置为以添加方式自动打开日志。这个习惯在危机时曾多次“解救”过我，而且还为我保留了大量案例处理的原始资料，这 10 多年来，已经积累了差不多 40 GB。这些资料为我后来归纳知识、回顾案例，提供了大量的宝贵信息。

老白一直在强调理解基本原理，并应用到实践中，但是有些时候，一个操作涉及的知识面太广了，我们可能无法考虑得那么全面，总会有一些遗漏。这时，好习惯就显得尤为重要了。这就好像开车一样，老白也喜欢在高速公路上开快车，不过在周围车比较多的时候，我会十分注意车距，因为我知道自己的车在 100 公里/小时的速度下，从刹车到静止需要 55 米左右的距离，因此我会尽可能和前车保持 50 米以上的距离。所以，保持好的开车习惯可以在关键时刻救你一命；同样，好的工作习惯，也往往会在关键时刻帮你“转危为安”。

在日常工作中，DBA 需要养成的好习惯有很多。

- 在去客户现场之前，要和客户通一个电话，了解工作的范围和重点，然后提前做一些功课，把与工作相关的知识要点准备好。每个人的能力都是有限的，知识面也都存在一些短板，做好充分的准备是对客户负责，也是对自己负责。
- 学会倾听，认真听取客户的想法、客户对工作的希望以及客户对项目的理解，这对于提高客户的满意度十分关键。无论技术多么优秀，任务完成得多么出色，不合客户的口味也是没用的。
- 操作之前，应打开各种能够记录操作过程的日志，比如 secureCRT 的会话日志。正如上述案例中提到的那样，这些日志可以挽回重大损失，有时也可以避免一些纠纷。
- 事先在自己的编辑器中写好所有的操作，最后再粘贴到虚拟终端上去执行。这个习惯一方面可以将每次操作的脚本都记录下来，另外一个方面也可以帮助我们在执行之前审阅脚本，最关键的是，可以有效地避免虚拟终端上的误操作。
- 碰到自己无法把控风险的操作，尽可能咨询一下其他人，或者尽早将其升级到二线或三线，千万不要轻易把自己置于未知的危险境地。当我们以团队方式工作时，没有必要一个人来承担所有的风险，只需要承担自己的那部分责任即可。
- 在离开客户现场前，要和相关负责人进行沟通，说明本次工作的内容，做了什么事情，有什么遗留问题，有什么后续建议。这样会让客户感觉你很专业，也很尽责。
- 完成一个项目后，应尽快写一份文档记录本次工作。就我个人而言，无论客户是否需要提交文档，我都会针对一些有趣的操作编写自己的文档。编写过程中，我会对本次操作进行总结，也会对自己做一次评审，看看操作过程中是否还存在需要改进的问题。

这里只列举了 7 条，实际上还有很多没有列出。其实，良好的习惯都是通过惨痛的教训换来的，每次失败和故障都应该成为我们改进的动力，记吃不记打的事情不应该在一个高度职业化的 DBA 身上多次发生。

第 15 章

DBA 分析思路的探讨

我们在碰到问题后该如何去分析，应从哪些方面入手去诊断和解决问题呢？这可能是每个DBA 都想学习的。其实这个问题并不复杂，我们可以先抛开 Oracle 数据库，来看看普通问题的分析方法。

在碰到问题时，我们首先会根据自身的经验去分析这个问题是属于哪一类的，接着会考虑在这一类问题上我们具备哪些经验，这些经验是否有助于我们分析问题。如果我们在这类问题上是有经验的，那么就可以根据经验进行思考和分析，直到找出解决方案；如果我们的经验不足以解决问题，那么可能就会向亲戚、朋友求助。

在互联网时代，我们也可能会考虑使用搜索引擎，查找相同或者类似的案例。但由于网络上的信息并不一定可靠，我们在搜索时，会花费大量的时间过滤信息，判断信息的正确性。使用搜索引擎最大的问题就是搜索结果并不一定正确，因此进行必要的判断是十分关键的。如果不加判断，直接引用，可能会带来严重的后果。

其实，DBA 在分析数据库问题时也离不开这些普遍规律，如果相关问题首次出现，可通过Metalink、谷歌以及百度进行搜索，或者向高手电话咨询。

本章的目的就是将老白这些年来总结的一些分析问题的思路分享给大家，不过每个 DBA 都有自己的思考习惯，因此老白希望大家能够借鉴这些思路，用以补充自己的不足，但不要生硬地全盘照搬。只有把这些考虑问题的方法和技巧完全融入到自己的思维体系中，本章的内容才能发挥真正的作用。一味生搬硬套，不懂得消化吸收，效果自然不好，甚至还可能导致十分严重的后果。

15.1 问题分析总路线图

抛开 Oracle 数据库，我们处理问题时，都有一个通用的路线图。首先要弄明白遇到了什么问题，问题的现状是什么，然后根据现状分析可能存在的处理路径，这个问题是归到哪一类的，以前是否碰到过，是否可以在自己的能力范围内解决。根据这些，我们会采取相应的处理措施。

作为 DBA，在碰到问题时，最先要做的事情同样也是了解问题的现状。如果我们在现场，就可以亲自去查看问题现状，而在其他时候可能只是听别人转述问题。这种情况下，我们接触到的第一手信息可能并不是问题的真正面目，因此不可尽信，需要从中分辨出真实的信息。这时，

如果手头有一些抓取信息的小脚本，那就最好不过了。通过这些脚本，我们可以抓取到最为原始、易于分析的数据。由于每个人的工作习惯不同，因此这些脚本的差别可能会很大。很多 DBA 都收集了大量高手或者大师的脚本，但是他们并不知道如何使用这些脚本，因此这些脚本虽然很好，但是没有什么用处。所以说，准备一些自己能够使用的脚本才是真正关键的。经过一段时间的沉淀，每个 DBA 都会积累一套使用起来得心应手的、适合自己的脚本，由于对这些脚本有十分深入的理解，因此通过这些脚本就可以完成绝大多数的分析工作。

不过有些时候，客户出于安全考虑，可能不允许将脚本上传到他们的服务器，因此我们还必须掌握一些简单的命令，用于信息的采集和分析。一般来说，这些命令必须简单易记，如果实在记不住，也可以记录在一个文本文件里，需要的时候拿出来查看一下（也就是说，常用命令不能太复杂，起码是可以一边看一边输入到客户系统中的命令，过于复杂的 SQL 不适合作为初步分析和信息采集的工具）。

无论如何，掌握第一手准确资料是最为关键的，分析问题的一个最为根本的前提就是所掌握的数据是正确的，否则分析不可能顺利进行。

现在我们把讨论范围集中到 Oracle 的问题分析上。如果某个 Oracle 数据库出现了问题，那么我们该如何分析呢？首先来看看我们是如何发现问题的。一般来说，数据库服务器故障主要表现在以下几个方面：

- 数据库无法登录；
- 数据库运行缓慢；
- 数据库出现坏块；
- 访问数据库出现报错，无法正常执行某些功能；
- 数据库整个挂起，无法执行 SQL；
- 某个 SQL 执行时挂起；
- CPU 使用率突然变大，甚至达到 100%；
- 某个业务功能突然变慢；
- 操作系统突然变慢。

虽然故障的现象千奇百怪，不过总结起来不外乎以下几种：

- 操作系统方面的故障；
- 数据库功能性故障；
- 数据库性能故障；
- SQL*Net 故障；
- SQL 性能故障。

由于每个系统都是不同的，所以对于不同的用户，经常要分析的问题种类也会有所不同，分析方法也存在差异。因此对于每个 DBA 来说，事先制定好一些分析预案是十分重要的。通过预案分析问题，可以大幅减少误判，从而提高问题的解决率。积累分析预案是每个 DBA 在日常学习、工作中必须要做的事情，只有这样，DBA 的工作能力才能不断地提升。也许很多 DBA 会说，我从来没有积累过预案，但是随着工作时间的增加，我的工作能力也在不断提升。实际上，我们

每时每刻都会碰到问题，都在分析和解决问题，每次解决问题后，我们或多或少都会有些记忆，下一次碰到类似的问题时，这些旧的记忆总是能够对我们有所帮助。因此，我们在不自觉地积累着处理预案。不过这种积累预案的方式效率较低，而且预案的质量也不高。如果每次处理问题后能够主动对处理过程进行提炼，并根据自己的知识推而广之，这样形成的预案质量就会大幅提高。下次遇到类似问题时，处理水平也会有所提高。

不过很不幸的是，绝大多数 DBA 的预案体系都不完备，而且其基本的案例积累也不足以完成问题的分析。在这种情况下，该如何进行分析呢？

由于系统出现的问题多种多样，因此查看系统资源、日志信息是十分重要的。很多 DBA 在分析故障时连日志都没仔细分析，就急着根据问题现象去谷歌、百度上搜索。虽然随着 DBA 工作经验的增加，用这种方法在大多数情况下也能找到合适的解决方案，但是存在的隐患同样也很大，经常会因为找到的处理方法不正确而导致问题无法解决，更为严重的是，如果操作过程存在风险或者不可逆，就有可能导致严重的事故。

作为一名 DBA，养成查看日志的习惯是十分重要的，因为 Oracle 的很多问题都会体现在日志文件中。这里所说的日志文件既包含 alert log 和各类 trace 文件，也包含操作系统和群集软件的日志。其中，操作系统和群集软件的日志是大多数 DBA 可能会忽视的，但实际上这些日志十分关键。另外，DBA 还应了解其维护的数据库系统、操作系统、集群软件以及与 Oracle 数据库相关的其他第三方软件日志文件的存放位置以及查找和分析的方法。对于系统故障、突然变慢、出现坏块、crs 无法正常启动等情况，检查操作系统日志是十分重要的。

除了检查日志以外，查看当前系统的资源使用情况也十分关键，很多故障都是由于系统资源出现问题导致的。因此，我们应重点检查 CPU、内存、I/O 和网络的情况。以前老白曾碰到过一个问题，在一套由 3 个异地节点组成的三向 STREAMS 复制环境中，主生产节点每天大概有 3000 万条记录要同步到华东和华中两个节点，华东、华中节点每天大约各有 200 万条左右的记录要同步到其他节点。有一次，主节点向华中同步的数据延时突然变得十分严重，而且在不断增大，已经达到几个小时。经过检查发现，STREAMS 环境正常，而捕获进程经常出现由于 UNBROWSE 消息数量过多而导致的流控，在华中节点的 APPLY 端，APPLY 进程大多数时间都比较空闲。通过 strmmon 工具分析发现，华中节点的网络传输速率只有 300 Kbit/s 左右，而一般情况下，如果出现较为严重的延时，网络传输流量的平均值不应该低于 600 Kbit/s。于是我们将这种情况初步定位为 PROPAGATION 进程方面的问题。在调整了一系列参数并重启该进程后，问题仍然没有解决，那么下一个怀疑的对象就是网络本身了。主系统到华中节点的网络采用了电信的 SDH，客户租用了一条 15 Mbit/s 的光纤通道。首先我们检查了操作系统的日志，没有发现网络报错的现象，于是又通过 netstat 命令检查网络的状况，也没有发现问题。接下来通过 ping 命令发现大小包的速度都在十几毫秒，属于正常范围。于是我们联系了运营商，运营商检查发现，这条专线没有问题，能够达到 15 Mbit/s 的传输峰值。既然运营商都已经确认网络没有问题，好像下一步就没有什么可用的方法了。不过我们好像还遗忘了一个最为简单的方法，于是我们停止了传播进程，然后使用 SFTP 去复制一个几千兆字节的大文件，在传输过程中我们发现网络确实存在问题，理论上 SFTP 的传输速度应该能够达到 1 Mbit/s 以上，而实际测试时，只能达到 400 Kbit/s 的速

度。既然运营商能够确保 15 Mbit/s 的传输带宽（也就是接近 2 MB/s，运营商的带宽单位是 bit），而实际的网络传输速度只能达到该值的 1/50，那么肯定有其他的应用在使用网络。由于缺乏足够的网络监控工具，我们只能对使用本网络的系统进行排查。经过排查发现，总部和华中节点间的一套电话视频会议系统的备用线路可以使用这条专线，于是我们马上和用户部门进行确认，得知由于今天视频电话会议系统主线路故障，所以临时切换到了备用线路。至此，问题终于明晰了，电话会议结束后，流复制也终于恢复了正常。

在这个例子中，我们采用了常用的排除法，首先从源头检查了 CAPTURE 进程，发现该进程是正常的，所在服务器的资源也比较充足，但这个进程经常出现流控现象，这说明该进程捕获的消息无法及时送出，从而导致流控。既然 CAPTURE 进程经常流控，那么是不是由于 APPLY 进程较慢导致的呢？于是我们马上检查了 APPLY 端，首先判断 APPLY 端的系统资源是否出现了瓶颈，经检查发现资源较为充足。接着我们又检查了 APPLY 服务进程的情况，发现该进程大多数情况下处于空闲状态，也就是说这个进程本身并无问题，而且足够空闲。既然排除了两端，那么重点怀疑的对象就是中间的 PROPAGATION 进程了。而经过检查发现每秒网络的传输量不足 400 Kbit/s，通过这一现象我们发现了一些问题，于是顺藤摸瓜，利用操作系统提供的检查工具，最终就找到了问题原因。

一般来说，从分析问题的总体思路来看，通过现有的知识以及后续的分析排除不可能的因素，从而逼近问题的本质是最为常见的分析方法。如果我们的知识较为丰富，就可以排除掉大多数不可能的路径，甚至直接定位到故障本身。不过如果从处理问题最为稳妥的方式来考虑，即使问题现象十分明显，也不要放弃必要的验证和排查，这样虽然麻烦一些，但是可以最大程度地避免偏差的出现。

15.2　普通故障的分析路线

这里所说的普通故障，是指一般性的、常见的日常维护问题，比如某个操作引起的错误，这种时候往往会出现“ORA-*XXXX*”这样的错误号。碰到这种情况，我们可能需要通过手头的技术资料或者 Metalink 网站，去查询这个 Oracle 错误所对应的含义。如果我们手头暂时没有技术参考手册，也没法登录 Metalink，并且面对的系统是 UNIX 系统，那么就可以使用系统的 oerr 工具，来查看这个 ORA 错误对应的含义以及可用的处理方式。oerr 命令的格式如下：

```
oerr ora <错误号>
```

比如，我们查看 ORA-1031 错误的含义，会得到如下结果：

```
[oracle@db ~]$ oerr ora 1031
01031, 00000, "insufficient privileges"
// *Cause: An attempt was made to change the current username or password
//         without the appropriate privilege. This error also occurs if
//         attempting to install a database without the necessary operating
//         system privileges.
//         When Trusted Oracle is configure in DBMS MAC, this error may occur
//         if the user was granted the necessary privilege at a higher label
//         than the current login.
```

```
// *Action: Ask the database administrator to perform the operation or grant
//          the required privileges.
//          For Trusted Oracle users getting this error although granted the
//          the appropriate privilege at a higher label, ask the database
//          administrator to regrant the privilege at the appropriate label.
```

从第一行来看，ORA-1031 表示权限不足。下面的 Cause 介绍了可能出现类似问题的原因，Action 介绍了需要采取的措施。

对于大多数 ORA-*XXXX* 问题来说，使用 oerr 工具可以让我们对问题有一个初步的了解，大多数问题也可以根据 Cause 和 Action 的内容得以解决。

如果 oerr 工具还不足以解决问题，那么 Metalink 就是一种较好的选择。登录 Metalink 后，直接输入 ORA-1031 进行查询，可以得到大量关于这个错误的相关技术资料，包括 Bug 的描述等。我们可以在 Metalink 上查找类似的案例，然后仔细阅读这些文章，寻找解决问题的方法。

但不幸的是，在某些情况下，我们可能无法通过 ORA-*XXXX* 错误找到问题，或是看不到明确的 ORA-*XXXX* 错误。这时该如何进行分析呢？

有时候，我们在进行某个操作时系统会突然出现 ORA-3113 之类的错误，然后会话就断开了。很可能在客户端出现 ORA-3113 错误前，后台已经出现了 ORA-600、ORA-7445 之类的错误。这种情况下，我们可以直接打开 alert log 文件，查看是否存在错误信息。如果找到了 ORA-600 或者 ORA-7445 的信息，下一步要做的就是找到错误相关的 trace 文件，并保存起来。然后在 Metalink 上根据错误参数查找相关的资料。一般来说，对于 ORA-600 错误，我们只需要查找其第一个参数对应的信息。比如，查找 ORA-600 [504]，输入查询条件后，屏幕上会显示出所有查找到的结果，如图 15-1 所示。由于 Oracle 的知识库十分庞大，有时我们查出的技术资料可能会非常多，如果要缩小查询范围，可以点击左边的导航条信息。比如，查找 Oracle 数据库的资料时，可以点击 Oracle Database Products 来进一步缩小显示资料的范围。

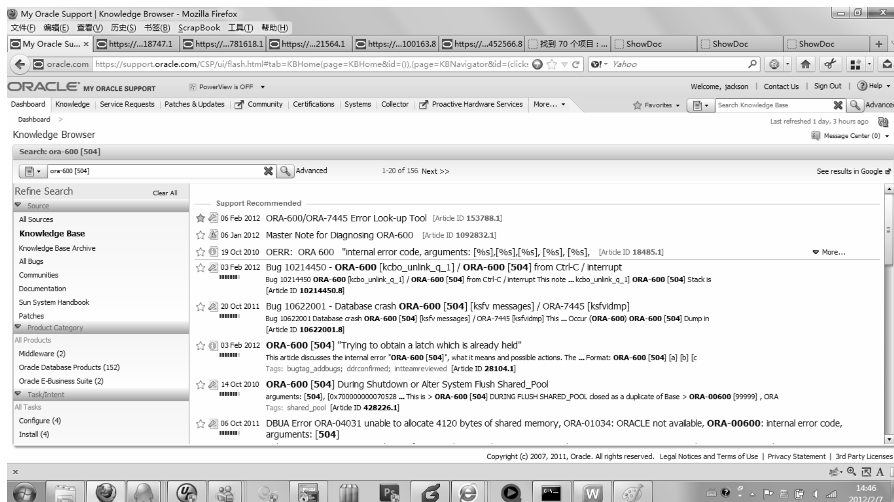


图　15-1

上面的查找操作是很简单的，但是令绝大多数 DBA 头痛的是，虽然查到了大量的技术资料，但还是不知道哪个文档才是有用的。在图 15-1 中，包含了 bug 10214450、bug 10622001 等查询结果，到底是哪个 Bug 引起了错误呢？这就需要我们认真查看刚才找到的 trace 文件，从中查找对应的进程、出错时正在执行的 SQL，通过这些内容再进行筛选，进一步查找是否存在相关的资料。

如果找到了类似的资料，那么就需要判断其中介绍的 Bug 或者案例是否和我们的情况相似。如果相似，下一步就要比对 trace 中的栈信息，如果栈信息也十分相似，那么这个文档所述的内容和系统当前出现的问题相同的可能性就很大了。接下来，我们就需要认真阅读这份文档，看看这类问题该如何处理。

下面通过一个示例来解释这个过程。比如，系统突然出现了 ORA-600 错误，导致某个业务模块无法正常工作：

```
Tue Apr 15 22:27:17 2008
Errors in file /opt/oracle/admin/orcl/udump/orcl_ora_5985.trc:
ORA-00600: internal error code, arguments: [qesmmCValStat4], [3], [1], [], [], [], [],
[]
```

首先，我们会根据 alert log 文件的内容找到 orcl_ora_5985.trc 这个文件，该文件的信息如下：

```
*** 2008-08-15 22:27:17.443
ksedmp: internal or fatal error
ORA-00600: internal error code, arguments: [qesmmCValStat4], [3], [1], [], [], [], [],
[]
Current SQL statement for this session:
```

于是我们到 Metalink 上去搜索，搜索结果如图 15-2 所示。

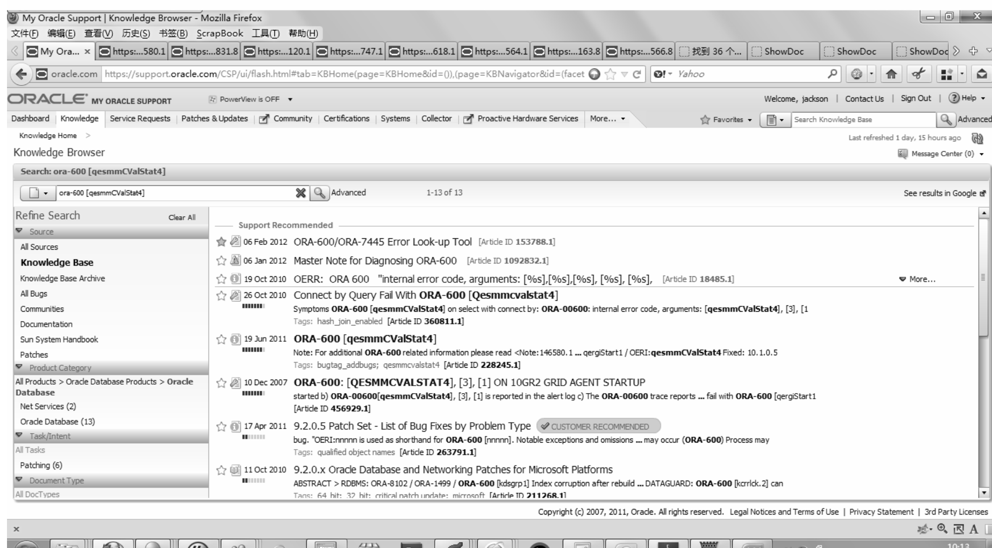


图 15-2

接下来，我们需要对查询结果进行排查和分析。幸运的是，这次查询只找到了 13 条记录，不过查出的数据较少也可能意味着这次查询找不到针对性的结果。

第一个条目是“Connect by Query Fail With ORA-600 [Qesmmcvalstat4] [ID 360811.1]”，我们检查了 trace 文件中的 Current SQL，这个语句就在 ORA-600 错误信息的后面，很容易找到。在 Current SQL 中，我们看到：

```
FROM prod.OFFERING_2_PRICE_PLAN O2PP, prod.PRICE_PLAN PP
         WHERE O2PP.PRICE_PLAN_CD = PP.PRICE_PLAN_CD)
CONNECT BY PRIOR A = C
 START WITH C IS NULL)
```

这里确实存在 CONNECT BY 子句，看样子我们这回比较幸运，很可能找到了问题的根源，不过还需要进一步确认。打开这个文档后，我们看到：

```
Applies to:
Oracle Server - Enterprise Edition - Version: 9.2.0.7 to 10.2.0.1 - Release: 9.2 to
10.2
Information in this document applies to any platform.
***Checked for relevance on 26-Oct-2010***
Symptoms
ORA-600 [qesmmCValStat4] on select with connect by:
ORA-00600: internal error code, arguments: [qesmmCValStat4], [3], [1]
Current SQL statement for this session:
select ...
... start with ... ... connect by prior ...
 The stack trace shows:
... qesmmCValidateStatus qesmmCStartWorkArea qerhjInitializeManagementComponents
qerhjFetch ...
Changes
You may have applied the patch for the Bug 4074409, but still get the same errors.
Cause
This is due to Bug 4926357, which is closed as duplicate of unpublished Bug 4401437.
Unpublished Bug 4401437 is fixed in versions 10.1.0.5, 10.2.0.2, 11.1 and above.
Solution
To implement the solution, please execute the following steps:
1. Check if there is an one-off patch for your platform and version (Patch 4401437)
or
2. Upgrade the database to a version where unpublished Bug 4401437 is fixed (see above).
or
3. For version 9.2: Set hash_join_enabled=false
or
4. Add a 'no_filtering' hint. Please note that this workaround should not be used in
version 9.2.0.7 due to Bug 4752555 'Wrong results from CONNECT BY query', which can
cause an incorrect number of rows to be returned.
```

不难看出，这个问题涉及的版本是 9.2.0.7 ~ 10.2.0.1，而当前系统的版本是 9.2.0.8，正好符合这个范围。距离问题又近了一步，继续往下看。这个 Bug 的成因是由于修复 bug 4074409 的补丁存在问题。经过确认，bug 4074409 的修复补丁的确包含在 9.2.0.8 版本中，在这一点上也是吻合的。下面我们就需要分析栈了，这份文档指出的 Bug 的典型栈是：

```
The stack trace shows:
... qesmmCValidateStatus qesmmCStartWorkArea qerhjInitializeManagementComponents
qerhjFetch ...
```

于是，我们分析 trace 文件中的栈信息：

```
ksedmp()+528          call     0000000000000000     000000000 ?
ksfdmp()+64           call     0000000000000000     000000003 ?
kgerinv()+400         call     0000000000000000     600000000004F280 ?
kgeasnmierr()+144     call     0000000000000000     600000000004F280 ?
[qesmmCValidateStatu call     0000000000000000     600000000004F280 ?
s()+560                                             600000000069A6C8 ?
                                                    40000000008ADEF0 ?
                                                    000000002 ? 000000004 ?
                                                    000000003 ? 000000004 ?
                                                    000000001 ?
$cold_qesmmCStartWo   call     0000000000000000     9FFFFFFF7F420DD0 ?
rkArea()+3296                                       000000001 ?
                                                    C000000000000795 ?
                                                    40000000015E7460 ?
                                                    0000081ED ?
                                                    6000000000524B48 ?
qerhjInitializeMana   call     0000000000000000     C000002043A0E440 ?
gementComponents()+                                 C00000000000122C ?
736                                                 4000000001B0B1B0 ?
                                                    00000802D ?
                                                    40000000008ADC10 ?
                                                    9FFFFFFF7F420DD0 ?
                                                    6000000000531E18 ?
                                                    6000000000530288 ?
qerhjFetch()+1392     call     qerhjInitializeMana  C000001FE84A2EE0 ?]
                               gementComponents()+  C0000000000052AB ?
                               736                  400000000224FAD0 ?
                                                    0000080ED ?
                                                    9FFFFFFFFFFF7A70 ?
                                                    6000000000531560 ?
rwsfcd()+240          call     <kernel>             C000001FE84A2EE0 ?
qeruaFetch()+448      call     <kernel>             9FFFFFFF7F41EE08 ?
qersoFetch()+1360     call     <kernel>             C000001FE84A2990 ?
qervwFetch()+320      call     c000001fb0cec338     C000001FE84A2908 ?
qercoFetch()+400      call     c000001fb0cec338     C000001FE84A28B0 ?
qerflFetchOutside()   call     c000001fb0cec338     C000001FE84A2858 ?
+272                                                000000000 ? 000000000 ?
$cold_qercbiFilterD   call     c000001fb0cec338     40000000012988D8 ?
ata()+1296                                          40000000012988D0 ?
qercbiFetch()+2448    call     c000001fb0cec338     C000001FB0CEC338 ?
rwsfcd()+240          call     qercbiFetch()+2448   C000001FB0CEC338 ?
qeruaFetch()+448      call     qercbiFetch()+2448   9FFFFFFF7F421228 ?
kpofrws()+288         call     qercbiFetch()+2448   C00000201942CA98 ?
opifch2()+3168        call     qercbiFetch()+2448   9FFFFFFF7F4F34A0 ?
opiall0()+6128        call     _etext_f()+23058430  9FFFFFFFFFFF8360 ?
kpoal8()+2272         call     0000000000000002     000000002 ?
opiodr()+3088         call     9ffffffffff9b98      9FFFFFFFFFFF9AD0 ?
ttcpip()+1888         call     9ffffffffff9d38      9FFFFFFFFFFF9E80 ?
```

15

```
opitsk()+1920        call     _etext_f()+23058430  6000000000052C40 ?
opiino()+2656        call     __text_start_f()+95  000000000 ? 000000000 ?
opiodr()+3088        call     __text_start_f()+95  60000000005E3CD8 ?
opidrv()+1088        call     9fffffffffffd7e8     9FFFFFFFFFFFD930 ?
sou2o()+48           call     9fffffffffffd7e8     9FFFFFFFFFFFE980 ?
main()+352           call     9fffffffffffd7e8     9FFFFFFFFFFFEF00 ?
main_opd_entry()+80  call     9fffffffffffd7e8     000000000 ?
```

我们注意到，方括号括起部分的栈信息和文档中的信息完全一致，这说明到目前为止，Bug 的吻合度是 100%，极有可能是这个问题引起的（为了保持本节的简洁，我重新编排了 CALL STACK 的信息，保留了 CALL STACK 中所有的 function call，但是删除了一些多余的信息，分析 CALL STACK 中的所有 function call 是否和自己的情况吻合是十分关键的）。那么该如何进一步验证呢？我们来看一下文档中的解决方案，也就是 SOLUTION 这一节。这里提出了 4 个解决方案，一是更新这个 Bug 的补丁；二是升级到已经修复了这个 Bug 的版本；三是设置 HASH_JOIN_ENABLED=FALSE，关闭 HASH JOIN；四是使用 no_filtering 提示。

接下来，我们将 SQL 完整地取出，首先在 SQL*Plus 下执行这条 SQL，确实每次执行都会报错，然后我们在会话级设置 HASH_JOIN_ENABLED=FALSE，错误就消失了。使用 no_filtering 测试，也能够成功执行。至此，我们基本上可以定位 ORA-600 错误和这个 Bug 有关了，那么下一步就是向领导申请停机更新补丁的时间，通过补丁彻底解决这个问题。

本节向大家介绍了普通 Oracle 问题的分析与解决思路。其实，解决问题的方法有很多种，每个 DBA 都有自己的思维方式，这里介绍的只是老白分析问题最常用的思路，并不要求所有的 DBA 都放弃自己的思考方式，采用和老白一样的方法。另外，每个 DBA 都有自己的知识库体系，在通过 Metalink 查找问题前，也可以先查找自己的知识库，不过前提是确保该知识库的正确性，否则可能会影响处理结果。在某些情况下，Metalink 可能无法访问，因此保留一个离线的 Metalink 知识库是十分必要的。早期的 Metalink 不是通过 HTTPS 协议连接的，可以通过工具去抓取离线数据，而现在已经无法抓取离线版本了，因此只能通过日常的积累，逐渐丰富自己的知识库。

15.3　性能问题的分析路线

在上一节我们讨论了普通的 Oracle 问题的分析思路，这一节重点来讨论性能问题的分析方法。性能问题的处理方式和普通故障有所不同，发现一般性能问题的途径包括以下几个方面。

- 客户投诉。客户系统出现性能问题，向 IT 部门投诉。一般来说，对于没有完善一线管理的系统，这是最可能的发现问题的途径，但因为客户对于系统性能下降的感受较为滞后，可能很难判断出某个操作端到端的响应时间是 200 毫秒还是 500 毫秒，因此，只要不出现性能大幅度下降，客户的主观感受较为迟钝。
- 监控报警。对于已经建立了维护基线，并进行全面一线监控的系统来说，可以通过监控报警来更早地发现系统存在的性能问题。监控系统对系统性能问题的敏感度远远高于客户，只要设置了合理的基线数据，那么监控系统就可以帮助我们及时发现系统出现的问题。这里需要指出的是，基线管理并不像想象的那么简单，不是随便设置一些参数就可

以实现自动监控的。我们必须设置合理的基线数据，如果基线设置过于宽大，那么就有可能把一些隐患掩盖掉，一些本应该报警的场景被直接过滤了，达不到防患于未然的目的；如果基线设置偏严格，那么可能出现大量的误报，从而导致运维团队风声鹤唳，最终大家对于监控报警直接忽略，从而失去了报警的意义。

- 事后分析。这也是我们经常碰到的、十分无奈的处理模式。由于问题已经发生，而且故障出现时为了尽快解决问题，往往缺乏必要的数据采集，因此我们能够掌握的分析数据极为有限，从而大幅降低了问题被准确定位的可能性。

对于不同的场景，我们所采取的分析路线图也会有所不同。在发生现场故障时，对于核心的企业级应用系统而言，分析和处理故障的时间极为有限，用户单位会有严格的解决时限。对于很多企业级用户来说，如果问题能够在 30 分钟内解决，那么此类故障不会影响维护团队的绩效；否则就可能会影响绩效。因此，30 分钟是每个现场支持工程师都无法回避的死限。如何在 30 分钟内解决问题往往是现场工程师首先面对的问题。所以说，现场故障处理的目标不是彻底搞清楚问题的根源在哪里，而是首先恢复系统的正常运作。

一般来说，如果系统出现了较为明显的性能问题，会体现在以下几个方面：

- 关键业务受到了影响；
- 系统整体性能不佳；
- 某些关键流程比较慢；
- 存在突发的负载增加；
- 某项系统资源存在瓶颈。

如果某些关键业务平时的性能良好，却在某个时间突然出现问题，那么就说明肯定有一些特殊的情况存在。很多 DBA 在碰到问题时总是急急忙忙地采取技术手段去分析和解决问题。尽管情况很紧急，但不进行任何调查研究就急于动手，是非常不好的习惯。老白这些年已经养成了一个很好的习惯，在碰到这类问题的时候，首先会让客户协助了解以下情况：

- 出问题前，或者前一天晚上，系统是否进行了升级，应用软件有无新模块或者补丁发布；
- 出问题前，或者前一天晚上，进行了哪些维护工作；
- 今天在系统上进行过哪些特殊的操作？比如，一些特殊的批处理业务，或者系统备份这类可能对系统有较大影响的操作；
- 今天业务部门是否进行了可能导致某个业务模块工作负载大幅度增加的活动。

这些问题完全可以交给其他人去做，而不需要自己一一确认。而且这项工作可以和分析工作同步进行，这样就能大幅加快问题分析的速度。

有一次，老白去给客户的系统做健康检查，刚到现场，就发现所有的维护人员都围在一个工位前，神色紧张。询问了一下才知道，今天一上班，系统的 CPU 使用率就达到了 100%，而平时这个系统的 CPU 使用率不足 50%，所有的业务部门都在报故障，系统很慢，已经严重影响了营业厅的业务。

我没有立即去分析问题，而是问他们前一天晚上进行了哪些维护操作，他们想了想说，没做过什么。后来，有一个 DBA 想起来了，昨天快下班的时候应用开发厂商让他给一张表加了一个

索引。于是我们立即生成了一份 STATSPACK 报告，我从报告中发现，确实存在几条开销很大的 SQL，从执行计划上看，这些 SQL 都使用了昨天新建的索引，而从业务的角度来看，使用这些索引是不合适的。于是我们马上对这张表和相关索引进行了分析，分析完成后，系统负载逐渐下降，10 多分钟后，CPU 负载也降低到了 50%以下。

老白通常把这种处理故障的过程称为“快捷方式”，就是通过经验和所掌握的信息，快速排除其他的可能性，从而定位问题的一种方法。快捷方式需要处理问题的人不仅技术能力较强，处理问题的经验很丰富，而且需要对系统比较了解，并且能够准确掌握相关的资讯。

如果使用快捷方式失败，那么就会浪费宝贵的分析时间。如果我们连续几次问题定位都失败了，那么就必须放弃使用快捷方式，而是采用常规的分析方法来进行分析。常规分析往往是一种自顶向下的分析方法，根据事先准备好的预案一点点地分析问题，最终找到问题，解决问题。

采用自底向上的方法是通过现象往本源方面推断，而采用自顶向下的方法是从本源向现象推断，根据我们现有的知识，按照一定的预案进行分析、排除，从而逼近问题点。整个诊断路线图就是不断排除、缩小范围的过程。虽然有时我们看到的现象是一样的，但是造成该现象的本源可能大相径庭，因此我们在分析时，如何按照正确的道路前行是十分关键的。首先来看图 15-3 所示的图例。

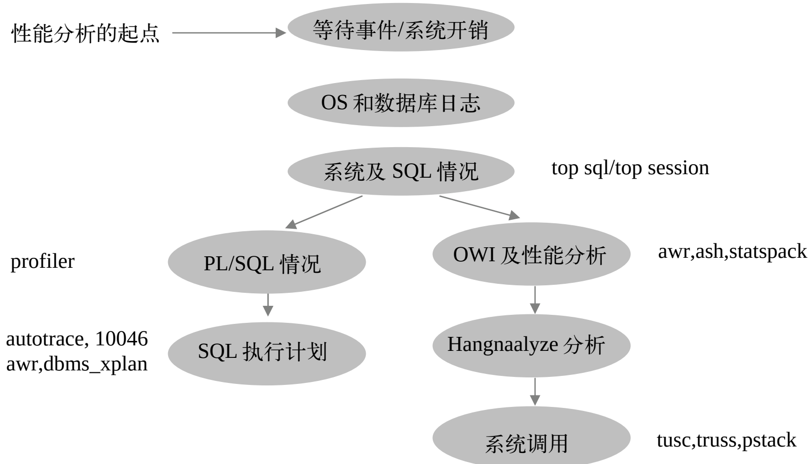


图　15-3

当我们碰到性能故障时，首先需要了解系统等待事件和系统资源开销的情况。查看操作系统是否出现了 CPU 或者内存不足以及换页的现象。如果是内存不足导致了换页，那么下一步就需要分析产生换页的原因，是物理内存确实不足，还是系统的某些问题导致了会话数量增加，从而耗尽了内存，又或者是某个 Bug 导致 Oracle 进程出现了内存泄漏，使一些进程消耗了过多的内存。这种情况下，杀掉部分消耗内存较大的进程可能就能够暂时缓解问题，但是找到问题的根本原因也是十分重要的。

除了内存、CPU 外，还必须检查网络，这也是我们经常忽视的问题。通过 netstat 等工具检

查网络是否存在问题，通过 ping 大数据包或者 ftp/sftp 等方式检查网络传输能力，这些都是排查网络问题的常用方法。

如果我们发现 CPU 使用率是 100%，而平时这个系统的 CPU 使用率总是小于 60%，那么就需要分析到底是什么原因导致了 CPU 使用率上升。top/topas 或者 glance 这类工具在这种分析中十分有用。我们可以通过 top 命令来查看是否存在某些大量消耗 CPU 资源的进程，如果发现了这些进程，就需要进一步分析这些进程当前的情况，然后通过 V$SESSION 视图来检查这些 Oracle 会话的作用，当前正在进行哪些操作，最后分析这些操作，找到问题的根源。

如果我们没有在 top 中发现异常，虽然 CPU 使用率很高，但是找不到明显的、开销很大的进程，那么就无法根据这条线索往下分析。因此，我们下一步就应该检查系统的各种日志，包括 Oracle 的 ALERT LOG 文件、操作系统的日志、集群日志等。如果有条件，还应该通知硬件、存储和网络维护部门检查服务器、网络和存储设备是否正常，是否存在明显的报错。

对于 DBA 来说，ALERT LOG 文件是检查的重点。在该文件中，我们应重点检查是否出现报错，比如 ORA-600、ORA-7445、ORA-4031、ORA-3113、ORA-3116、IPC SEND/RECEIVE TIMEOUT 等，如果发现了报错信息，就应该进一步判断这些问题是否和系统变慢有关（很可能绝大多数 ALERT LOG 的报错都和系统变慢无关，因此我们也不能草木皆兵，必须认真分析，才能得出结论）。

如果没有在 ALERT LOG 文件中发现什么有价值的信息，那么接下来就需要通过 OWI 工具来检查系统目前的状态，查看 V$SESSION_WAIT 中的主要等待事件是否存在异常，和我们平时看到的等待事件有无不同。经常管理这个系统的 DBA 可能很了解本系统平时的情况，但一个刚接触这套系统的人，并不清楚这套系统平时是什么样的，而 V$SESSION_WAIT 中的等待事件又不存在很严格的正常情况和非正常情况，那么可能就需要花费更多的精力对大量的等待事件进行分析和排查，这样就增加了分析问题的时间。因此，在维护过程中，记录系统的一些正常指标是十分关键的。我们可以把下列 SQL 运行后的结果保存起来，作为基线。

```
select count(*),event from v$session_wait group by event order by count(*)
```

一旦系统出现问题，我们在进行分析时，可以将查询结果和这个基线数据进行比较，这样就很容易发现其中存在的问题了。

另外还需要注意当前正在执行的 SQL，很多情况下，系统出现的问题并不是由某一个消耗了过多资源的会话引起的，而是由于大量的并发小 SQL 导致的。有些平时开销很小的 SQL 往往会被我们忽视，但当这些 SQL 的平均执行开销增加 30%，或者并发量增加 30%时，就有可能给系统带来致命的打击。

在 10*g* 版本中，我们可以通过 SQL_ID 来分析是否存在一些导致系统问题的 SQL 语句。在 9*i* 或者更早的版本中，我们可以使用 SQL 的 HASH_VALUE 来代替 SQL_ID。

```
select count(*),SQL_ID from v$session_wait group by SQL_ID order by count(*)
```

但是如果我们遇到的是一套代码很不规范的系统，没有使用绑定变量，那么就无法通过 SQL_ID 来查找存在问题的 SQL 了。这种情况下，一些抓取 TOP SQL 的工具也会失去作用，因

此，ASH 报告和 AWR 报告将是分析问题的有力工具。

如果我们在故障现场，而正好系统安装并启用了 DB CONSOLE 或者 GRID CONTROL，那么 EM 优秀的性能分析和 SQL 优化工具会带来很大的帮助。这里，登录 EM 并调出顶级活动管理页面，如图 15-4 所示。

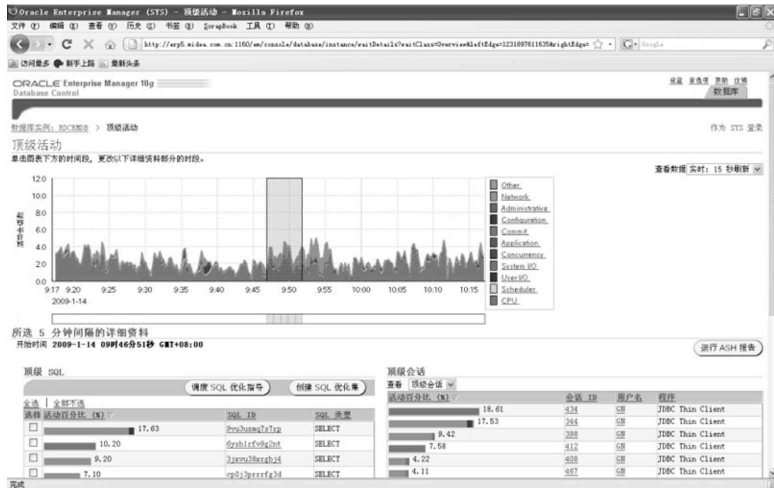


图　15-4

通过顶级活动的曲线图，我们可以看到系统在某个时段出现了高峰，拖动时间窗口到某个位置，在本页面的下方就会显示 TOP SQL 和 TOP SESSION 的情况。如果发现某条 SQL 可能存在问题，只需点击这条 SQL，就可以对该 SQL 进行分析及在线优化，如图 15-5 和图 15-6 所示。



图　15-5

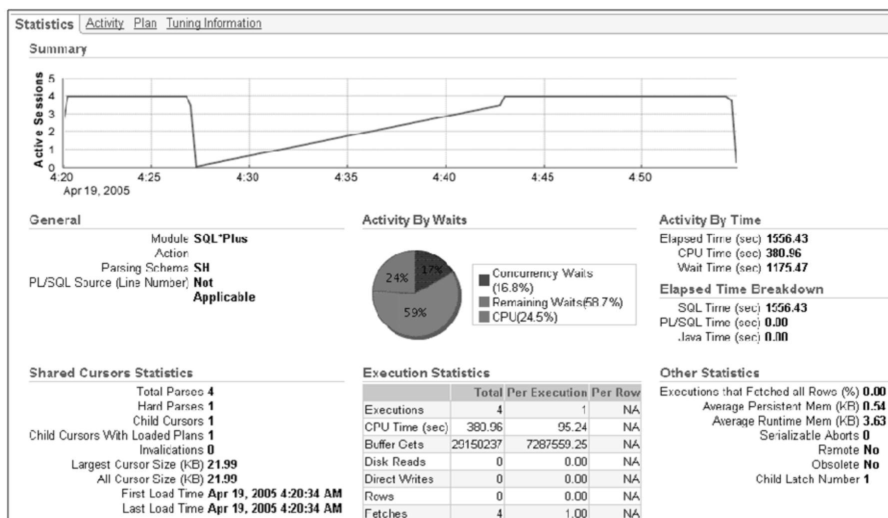


图 15-6

我们还可以直接在 EM 中分析执行计划，如图 15-7 所示。

ORACLE Enterprise Manager 10g
Database Control

设置 首选项 帮助 注销
数据库

数据库实例: RDCRMDB > 指导中心 > SQL 优化结果:TASK_23024 > 以下 SQL ID 的建议案:3gt1b5mq4shaq > 原解释计划

作为 SYS 登录

原解释计划

指示由 SQL 优化指导对原始计划进行的调整
以下是正在优化的 SQL 语句的原始解释计划。

全部展开 | 全部隐藏

操作	行 ID	对象	对象类型	顺序	行	大小 (KB)	成本	时间 (秒)	CPU 成本	I/O 成本
SELECT STATEMENT	0			11	1	3.173	2437	30	26921054	2429
COUNT STOPKEY	1			10						
FILTER	2			9						
VIEW	3			7	1	3.173	2432	30	26761630	2424
SORT ORDER BY	4			6	1	0.768	2432	30	26761630	2424
NESTED LOOPS ANTI	5			5	1	0.768	2431	30	23330950	2424
TABLE ACCESS BY INDEX ROWID	6	DRP_DIFFPROCBILL_HEADER	TABLE	2	960	691.875	506	7	6105765	504
INDEX RANGE SCAN	7	INDEX_DRP_DIFF_ENTID_ORGID_C	INDEX	1	998		7	1	249650	7
TABLE ACCESS BY INDEX ROWID	8	PRINT_BILL_CTRL	TABLE	4	35945	1,684.922	2	1	17943	2
INDEX RANGE SCAN	9	PRINTBILLCTRLBILLTYPE	INDEX	3	5		1	1	8971	1
TABLE ACCESS FULL	10	CPCORG	TABLE	8	1	0.012	5	1	159424	5

图 15-7

如果我们自身的分析能力不足，或者不想伤害更多的脑细胞，那么就可以直接调用 SQL 优化助手进行分析，运气好的话，能够马上得到优化方案，并直接在 EM 中提交优化方案，进行现场实施。

不过有可能我们并没有那么幸运，也许系统根本就没有配置 DB CONSOLE，或者 DB CONSOLE 无法使用，那么我们就只能依靠自己的双手，多费点脑子了。这时 ASH 报告和 AWR 报告就十分有用了。

很多朋友在分析问题时很喜欢套用以往的经验，这样确实可以缩小分析的范围，达到事半功倍的效果。但如果我们一味地墨守成规，那么经验不但不会帮助我们，还会带来很大的负面作用。

因为尽管问题的表象可能很相近，但是其根源却大不相同。比如，下面这两个案例的表象都是 CPU 使用率 100%，不过经过分析发现，是由截然不同的原因导致的。首先我们来看第一个案例：

```
HP-UX hpux-01 B.11.11 U 9000/800    07/25/05
12:47:04    %usr    %sys    %wio   %idle
12:47:09      31      69       0       0
12:47:14      20      80       0       0
12:47:19      11      89       0       0
12:47:24      22      78       0       0
12:47:29      23      77       0       0
12:47:34      14      86       0       0
12:47:39       7      93       0       0
12:47:44      10      90       0       0
```

从数据上看，CPU 的 IDLE 为 0，SYS CPU 的比例很高。这种情况下，USER 的 CPU 使用率不高，因此不太可能是由于大量 SQL 扫描数据导致的。以老白的经验来看，有两种情况可能性最大，在系统换页或者 LATCH SPIN 很严重时，都会出现大量的 SYS CPU 消耗。这里，WIO 的数据为 0，而如果换页大量出现时，WIO 不可能为 0，因此我们更倾向于 LATCH SPIN。通过 vmstat 检查很快就能排除系统换页的可能性。于是马上采集一个 STATSPACK 报告，果然 LATCH FREE 等待出现在 top 5 event 里，主要体现在 LIBRARY CACHE 和 SHARED POOL 方面。

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                  % Total
Event                                            Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file sequential read                         1,335,960       9,089    45.46
library cache pin                                   2,477       7,008    35.05
CPU time                                                        1,606     8.03
library cache load lock                               684         969     4.85
latch free                                         88,088         702     3.51
```

可以看出，LIBRARY CACHE PIN 排在第二位，占等待事件的 35.05%，从后面的明细情况来看：

```
                                                                Avg
                                                   Total Wait   wait    Waits
Event                           Waits   Timeouts   Time (s)    (ms)     /txn
---------------------------- ------------ ---------- ---------- ------ --------
db file sequential read         1,335,960          0      9,089      7    114.8
library cache pin                   2,477      2,316      7,008   2829      0.2
```

LIBRARY CACHE PIN 的平均等待时间为 2829 毫秒，超过正常水平数百倍（在一般情况下，这个等待为几毫秒），这是很典型的共享池冲突现象。同时在 ALERT LOG 文件中也出现了大量的 ORA-4031 报错信息：

```
ORA-04031: 无法分配 4200 字节的共享内存 ("shared pool","select a.customer_id,a.deale...",
"library cache","kkslpkp - literal info.")
```

通过 4031 分析，我们发现共享池的空闲比例还很高，于是决定马上暂停部分非 7×24 小时的应用，然后刷新共享池，问题就解决了。

另外一个案例中，系统的 CPU 也很忙，不过 SYS 的 CPU 比例并不高，只有 5%左右，相关的 AWR 报告如下：

```
Event                          Waits    Time(s) Avg Wait(ms) %  Total Call Time  Wait Class
CPU time                               195,685                  31.3
db file sequential read   6,424,327  55,647        9             8.9              User I/O
log file sync             1,546,901   9,521        6             1.5                Commit
db file parallel write    5,226,356   4,519        1              .7            System I/O
gc buffer busy              620,869   3,857        6              .6               Cluster
```

从报告上看，并没有什么异常。那到底是什么原因呢？马上用 topas 命令检查是否存在高开销的进程，并没有发现特殊的进程。于是，我通过 SQL_ID 在 V$SESSION 中对激活的会话进行了分组，发现居然有 200 多个活跃会话在执行相同的一条 SQL。接下来，通过 awrsqrpt 比对了这条 SQL 前几天的情况，发现执行计划没变，CPU 开销也没变，不过平均每次执行时间比以前长了七八倍。再回头查看 AWR 报告，发现 DB FILE SEQUENTIAL READ 的平均等待时间平时是 5～6 毫秒，而现在是 9 毫秒，这有些不正常。经过分析和多方了解，我发现有一张表使用了一个新的分区，而这个分区建在了一个新加的 RAID 组上。这个 RAID 组本来是用来备份文件的，所以做了 RAID 5，而且磁盘的数量也较少，只有 3 块，由于其他磁盘组空间较为紧张，才把新的数据文件建在了这个 VG 上。通过对新建的文件进行分析，我发现访问这个文件的 I/O 性能确实比其他文件要低很多。于是我们决定晚上把这个数据文件迁回性能较好的 VG。处理后，第二天系统就恢复正常了。

通过上面这两个表象极为相似但处理过程截然不同的案例，我们应该明确一个道理，那就是不要轻易下结论，按部就班地按照 CHECKLIST 一点点分析，可能是最快的途径。想要跳过分析环节，直接得出结论，在大多数情况下，可能会欲速则不达。

15.4　SQL 语句的分析路线

SQL 优化是优化工作中最为耗时的部分，很多 DBA 都觉得 SQL 优化是十分高深的工作，不敢涉足。实际上，只要掌握了基本思路，SQL 优化并不像想象的那么难。老白一直认为 SQL 优化是一件体力活。

SQL 优化工作实际上分为两部分，首先要分辨出哪些 SQL 需要优化，然后针对这些 SQL 进行优化分析。对于应用优化项目而言，一般包含以下几方面的工作：

- 查找 TOP SQL；
- 分析 SQL 对系统的影响；
- 分析 SQL 的优化方法；
- 制订优化计划；
- 实施优化操作；
- 评估优化效果。

查找 TOP SQL 的方法有很多，现在大家最常用的是通过分析 AWR 报告或者 V$SQLAREA

来查找。我们一般会通过某个时间窗口来查找这段时间内的 TOP SQL，这种查找方式没有问题，不过要注意的是，不同的时间段中，如果系统执行的 SQL 差别较大，那么就需要检查所有的关键时间段，查找可能存在的 TOP SQL。老白经常使用的查找 TOP SQL 方法包括：

- AWR/STATSPACK/ADDM/ASH 报告；
- EM ADDM 分析；
- SQLA（来自 Metalink 的工具）；
- VSQL/VSQLAREA；
- Oracle 9*i* SQLANALYZER；
- Oracle 9*i* EM TOP SQL。

找到 TOP SQL 后，我们不能埋头就开始进行 SQL 的优化，而是要首先进行甄别，剔除那些优化价值不大的 SQL。那么哪些 SQL 才是优化价值不大的 SQL 呢？

首先，一次性执行的 SQL，特别是用 SQL*Plus 或者 PL/SQL DEVELOPER 执行的 SQL，这些 SQL 往往是由某些维护人员或者业务人员手动执行的，其执行频率很低，而且经常会改变，优化的价值不大。

其次是一些对 OLTP 系统性能影响较小的批处理业务，这些 SQL 虽然周期性运行，但执行频率很低，只要这些 SQL 没有严重影响到 OLTP 业务的性能，就可以将其优先级设置为较低，不在优化初期重点分析。

最后是一些可以安排在夜间业务低谷时运行的批处理作业，这些 SQL 往往是一些十分复杂的统计操作，优化难度较大，只要避免其在白天业务繁忙时段运行基本上就能够解决问题了，在优化初期先不要去啃这种硬骨头。

在筛选过程中，还需要为这些 SQL 的优化工作安排优先级。如果我们不是做一个优化项目，而是应对一个突发事件，那么就应该尽可能先选择一些较为简单且分析难度不大的 SQL 进行优化，最后再去处理那些十分复杂的 SQL。但如果我们是在一个优化项目中进行 SQL 优化，那么对系统性能影响越大的 SQL，其优先级就越高。

在分析和优化 SQL 时，要注意成本的控制，尽可能选择性价比最高的优化方案，而不是性能最佳的优化方案。如果有两种优化方案，第一种是通过添加一个索引来解决问题，而第二种需要修改 Java 代码。这种情况下，第一种方案能够减少 80%的资源开销，而第二种方案可以减少 95%的资源开销，那么我们该选择哪种优化方案呢？显然，第一种优化方案应该是我们的首选，当然在条件允许的情况下，比如软件要进行升级，那么第二种方案也有可能实现。

除了成本外，还要考虑风险，尽可能规避风险也是优化方案选择中必须考虑的因素。我们应该尽可能选择风险较小的方案，避开风险系数较高的方案。

SQL 优化的最佳途径并不仅仅是改写 SQL，实际上，优化的方法有很多种，老白归纳了一些最为常见的 SQL 优化方法：

- 调整索引；
- 调整执行计划；
- 优化相关表的存储结构；

- 数据归档；
- 表和索引分析策略调整；
- 调整 SQL 执行时间窗口；
- 限制数据查询范围；
- 修改 SQL。

调整索引是优化成本最小的解决方案，在老白参与过的优化项目中，通过索引优化的 SQL 大约占 50%～60%，超过一半的 SQL 都是可以通过索引进行优化的。

除了索引外，还有些 SQL 的执行计划存在问题，通过调整执行计划来优化 SQL，减小其开销，也是十分常见的方法。

优化相关表的存储结构包含十分广泛的含义，最简单的是调整表的存储参数，还有就是将普通表改为分区表或者 HASH CLUSTER、IOT 等结构，当然，也包含了通过 MOVE 或者 SHRINK 操作减少表的碎片。通过调整表的存储结构可以减少 SQL 的执行开销，从而达到优化的目的。这种情况还存在一个特例，就是通过调整表中记录的顺序，减少索引的 CLUSTER FACTOR 值，从而达到降低索引范围扫描成本的目的。在《Oracle DBA 优化日记》这本书中，老白曾介绍过一个类似的案例，有兴趣的朋友可以查阅一下。

数据归档也是最有效的 SQL 优化方案之一，不过这个方法常常被忽视。有些 SQL 经常要进行全表扫描或者分区扫描，对于这类 SQL 来说，普通的优化方案很难奏效，如果表中的数据是有时限性的，那么设计历史数据查询功能，定期对生产数据进行归档，就可以控制这些 SQL 的开销了。

如果我们在一张包含数千万条记录的大表中，仅仅查询其中某几天的数据，那么通过索引就可以很快地完成查询，但是如果我们要查询半年的数据，那么就只能进行全表扫描了。对大表进行全表扫描的成本很高，并且经常会严重影响系统的性能。如果碰到这种情况，我们该如何处理呢？由于客户查询范围的不确定性，我们很难强制指定通过索引来访问这张表，不过如果可以限定最多只能够查询 3 个月的数据，那么所有的查询就都可以通过索引来完成了，这样就能避免出现全表扫描。虽然这种限制对业务人员来说会有所不便，但是避免了性能问题。如果我们在应用设计上考虑得更完善一些，提供一些能够归并多次查询结果的小工具，那么对业务人员的不便也就降到了最小。这一点老白还是和香港人学的。十多年前，我曾为香港某公司外包开发过一个系统，这个系统的每个查询界面上都设计了“归并查询”和“归并统计结果”这样的按钮，通过这些按钮可以将一系列类似的查询结果进行归并，并可以很方便地调出归并后的结果。

修改 SQL 是我们最不愿意做的事情，不过这也是我们必须直面的问题。当上述方法都失效时，修改 SQL 就不可避免了。

老白讲了半天，还没有切入正题，大家可能有些着急了。其实不然，SQL 优化技术实际上并不仅仅是抓住一条 SQL，去分析执行计划这样的体力活，如果能够四两拨千斤，谁还会去傻费力气呢？但既然是做 SQL 优化，就肯定免不了要干这种苦活、累活。很多 DBA 对于分析 SQL 的执行计划感到十分头痛，其实只要掌握了基本的方法，这项工作也并不是什么难事，顶多算是比较费力的事情而已。

实际上，SQL 优化最为关键的因素主要体现在三个方面：一是多表连接的顺序，二是两个表连接的方式，三是单表访问的路径。无论多么复杂的多表连接，最终都可以落实到这三点上。实际上，如果分析过 10053 TRACE 文件就会发现，10053 事件选择执行计划时也是从这三个方面来考虑的。

在分析复杂 SQL 时，首先要分析 SQL 中每个表的过滤条件，确定每个单表的最佳访问路径。表的访问路径大体可分为全表扫描、索引唯一性扫描、索引范围扫描、快速全索引扫描、索引跳跃式扫描、分区扫描等。

对于每张单表，首先要根据表上面的过滤条件确定每张表经过过滤条件后可能产生的结果集的大小。然后根据结果集和表的大小的比较，我们就可以选择出合适的单表访问路径。这里，通过对比表的记录数和过滤条件过滤后的结果集的大小，就能够作出判断了。

对于多表连接，确定表的连接顺序是最为关键的，表连接顺序的选择要素是尽可能多地过滤掉无效的记录。因此过滤条件较多的表，最终过滤效果最好的表会排在最前面。

```
SELECT
  PKG _SP_SEQ.F_E_MP_P_SNAPID,
  B.PRC_TACTIC_SNAP_ID,
  ......

FROM
  BF_BILL_PARA A,
  CONSPRC_SNAP B,
  CONS_SNAP C
WHERE
A.TARIFF_ID                   = B.PRC_ID
AND TRUNC(A. CHG_DATE)        <= TRUNC(:B5 )
AND A.SP_ID                   = B.SP_ID
AND A.CONS_ID                 = C .CONS_ID
AND B.CALC_ID                 = C.CALC_ID
AND A.APP_NO                  = B.RELA_APP_NO
AND C.APP_CODE = :B4
AND B.ORG_NO                  = :B3
AND C.ORG_NO                  = :B3
AND A.CALC_ID                 = -1
AND NVL(A.CHG_DESC, '02')     <> '01'
AND NVL(A.CHG_TYP E, '01') <> '02'
```

这条 SQL 在 A、B、C 三张表上都有过滤条件，另外，A 和 B、B 和 C、A 和 C 都有关联条件。这种情况是十分复杂的，因为这三张表都可能作为第一张关联表。如果我们要确定关联关系，就需要比较这几张表上的过滤条件，判断哪个条件更强。通过分析每张表上的所有过滤条件过滤后的结果集的大小，就可以找出最佳的那张表。在这个案例中是 C 表，这张表上的 APP_NO + ORG_NO 是很强的过滤条件，ORG_NO 是分区主键，经过过滤后，从一张 2000 多万条记录的表中筛选出了 20 多条记录。确定了驱动表后，就要考虑 C 表的访问路径了，到底是全表扫描好，还是按索引扫描好。这是一张很大的表，有 2000 多万条记录，全表扫描肯定是不可取的。而符合 APP_NO=:B4 条件的记录大约有 200 多条，符合 APP_NO=:B4 AND ORG_NO=:B3 组合条件的

记录大约有 20 多条。因此可以确定，如果存在 APP_NO + ORG_NO 的索引，对于这条 SQL 来说，是最优的。

由于 C 表过滤后只有 20 多条记录，因此嵌套循环（nested loop）可能是较好的选择，C 作为嵌套循环的驱动表。选定了驱动表后，就需要分析 C 表先和哪张表连接比较好。选择的原则是，能够尽可能多地过滤掉数据，连接后返回结果集较小的优先考虑。这种情况下，我们可以通过改写 SQL，来判断符合条件的数据的数量，比如，先来判断 C+A 连接的方式：

```
SELECT
  Count(*)
FROM
  BF_BILL_PARA A,
  CONS_SNAP C
WHERE
TRUNC(A. CHG_DATE)            <= TRUNC(:B5 )
AND A.CONS_ID                 = C .CONS_ID
AND C.APP_CODE = :B4
AND C.ORG_NO                  = :B3
AND A.CALC_ID                 = -1
AND NVL(A.CHG_DESC, '02')     <> '01'
AND NVL(A.CHG_TYP E, '01')    <> '02'
```

同样，我们也可以分析 C+B 的情况，通过比较，确定一个较好的连接顺序。这样就找到了表连接的顺序和表连接的方式。

也许我们会面临更为复杂的 SQL，包含了 INLINE View、子查询、CONNECT BY 的树状查询等，不过大多数查询最终都可以改写为等价的表连接的方式。这种情况下，需要首先将 SQL 中的这些内容改写为等价的表连接，然后再进行上述案例中所做的分析。仅此而已，看上去是不是很简单？实际上，SQL 优化也确实很简单，其原理和方法都是几页纸就可以说清楚的。但是真正实施起来可能就没有那么简单了，真正的 SQL 优化高手都是实践出来的，因此看再多的书，也不如自己亲手去优化几个 SQL。

想要成为 SQL 优化高手的朋友们，看了本节，你们是不是也想去一试身手了呢？如果是这样，那还犹豫什么呢，赶快行动起来吧。也许过上一年半载，你也可以骄傲地和别人说，SQL 优化没什么神秘的，无他，唯手熟耳。

15.5 利用你知道的原理缩小问题的范围

在诊断问题时最麻烦的就是某些问题涉及的知识点很多，这种情况总是让我们无从下手。此时，知识面的宽窄就显得十分重要了。最近这几年，总是有网友问我一些底层的核心问题。遇到这种情况，我会很耐心地建议他们先打好基础，在初学阶段不宜过于深入地研究某些问题，因为 Oracle 的知识浩如烟海，在刚开始学习时，我们应该花大量的时间扩大知识面，一味地死抠某些核心问题，往往得不偿失。

如果我们的知识面够广，那么碰到一些故障时，就可以先利用自己所掌握的知识，缩小分析范围，或者将分析要点根据可能性高低排序，进行重点排查，这样就可以加快分析的速度，提高

解决问题的成功率。

比如，一个系统的某项业务在每个星期六早上总是会短暂地挂起大概一分钟左右，而且几乎每次挂起的时间都基本一样。这种情况下，我们首先会想到是不是当时系统资源出现了不足，但通过操作系统监控进行分析后，排除了这种可能性。此外，操作系统日志也没有任何报错信息，这就说明系统本身并不存在问题。那么下一个需要检查的就是 ALERT LOG 文件了，但该文件也没有任何问题。然而，在分析 ASH 数据时，我们发现当时被挂起的应用都在等待 CURSOR:PIN S WAIT ON X 事件。这时，就需要我们利用已掌握的知识来缩小问题的分析范围了。CURSOR:PIN S WAIT ON X 等待事件和共享池相关，如果我们对这个知识点理解得更深入一些，就会发现这个等待事件从 Oracle 10*g* 起才出现，并且和互斥问题有关，是会话执行 SQL 时访问游标产生的等待。如果要分析当前的互斥等待情况，就需要进行一次 SYSTEM STATE DUMP 操作，但由于我们是在分析以前的故障，因此无法进行采集。不过我们可以把分析范围限定在共享池、SGA 方面，接下来就需要分析共享池或者 SGA 方面是否存在问题。通过 V$SGA_RESIZE_OPS 视图，我们发现在故障出现前后，系统出现了大量的 RESIZE 操作，主要是共享池和 DB Cache 之间的相互缩减。

接下来，我们继续分析这个问题。故障发生时，系统并不忙，而是处于从比较空闲转为较忙的中间点上，但在半小时后，就进入高峰时段了。因此这个时间段也是 SGA RESIZE 操作十分频繁的时段。至此，我们似乎已经找到了问题的关键点，这个故障可能是由 SGA RESIZE 操作引起的。但如果我们对相关知识点十分了解的话，就会发现一个疑点，这个故障总是发生在固定的时间点，而如果仅仅是由于 SGA RESIZE 操作导致的故障，时间上可能会有些差别，甚至有时差异会相当大。于是我们就想到了另外一个问题，到底什么才会导致周期性的故障呢？很可能这个故障和某个定时任务有关。接下来，就需要检查所有的调度（schedule）、作业（job）和操作系统的定时任务（crontab），凡是和共享池抖动有关的定时任务都是需要重点分析的对象。在分析过程中，我们仅仅发现了一个在这个时间点执行且可能和共享池有关的作业，该作业在每周六都会对一个触发器进行重新编译，这是因为该触发器涉及了一项较为复杂的统计业务，几乎每周触发器的内容都会进行一些微调。这个触发器涉及的表并不是出现故障的那张表，不过通过共享池和游标的相关知识我们知道，一旦触发器修改，那么这张表相关的游标都会处于 INVALID 状态，下次执行时就需要重新编译。这也可能导致共享池的争用加剧，而这时候正是业务量增长比较迅速的时间段，此时出现 SGA RESIZE 操作就是有可能的了。于是我们将这个作业的执行时间提早了 1 小时，故障就消失了。

在这个案例中，我们不断地使用已掌握的知识缩小了分析范围，并逼近问题的根源。实际上，最终这个案例也没有找到根本原因。不过已经足够让我们解决这个问题了。其实在我们遇到的绝大多数案例中，都存在这样的问题，尽管最后我们能解决问题，却无法弄明白问题的最终原因。

想要利用所掌握的知识缩小分析范围，就必须对所分析问题涉及的知识点掌握得十分准确，但并不需要非常深入。在上述案例中，只要我们能从 CURSOR:PIN S WAIT X 等待联想到 SGA RESIZE 操作就已经足够了，并不需要了解更深入的知识，这种情况下，需要的是知识的广度而不是深度。如果我们从这个等待事件入手，深入分析游标、互斥这些问题，那么就很可能要绕一

个大弯才能和某个作业挂上钩，甚至在我们深入研究了游标的底层核心问题后，会发现离问题的真相越来越远了。

在本节的最后，老白要重申的是，知识的广度远比深度重要，先有广度，再有深度才是真正的学习之道。如果一头扎下去，深入研究底层核心问题，很久后抬头一看，可能会发现自己其实并没掌握什么有用的知识。

15.6　关闭问题的条件

发现问题、分析问题、解决问题、总结问题，这是处理问题的一条正常路径。我们在分析故障时，有可能当时并不能真正了解问题的根源，有时候问题虽然解决了，但是并没有找到导致问题的真正原因。很多 DBA 处理完案例后并没有总结的习惯，而是应付完现场工作就草草了事了，甚至如果系统经过重启后问题解决了，也就不再过问其他的事情了。

这种习惯其实是很不好的。一方面问题的原因没找到，下一次发生类似故障的几率很大；另一方面，这也不利于自身能力的提高。对于每个问题，只要还没有找到真正的原因，那么这个问题就应该还处于开启状态，不能关闭。这也就意味着我们还必须继续分析这个问题，直到有充分的理由可以关闭该问题为止。

以老白的经验，一个问题如果在最近的两个月内没有重现，那么这个问题的重要性也就没那么高了，而且如果两个月内我们都没能关闭这个问题，就说明该问题可能不在我们处理的能力范围之内，或者缺少某些诊断资源，无法进一步分析。因此老白通常将问题强制关闭的时间设定为两个月，对于尚未关闭的问题，在两个月内一定要进一步分析，哪怕事情再忙，也要抽出时间来完成。如果缺少数据，可能还需要客户协助补充。但如果某个问题在两个月后，还没有任何实质性的进展，那么就可以暂时关闭该问题了，因为精力和资源有限，让那么多问题都处于开启状态，可能会让我们疲于奔命。表 15-1 是老白使用的问题登记表。

表　15-1

问题类别	客户名称	发生时间	关闭时间	关闭类别	故障描述	故障分析	相关资料位置

对于那些已关闭的问题，我们可以在半年或者一年后进行回顾，因为在这段时间里可能处理了一些类似的案例或者某方面的能力得到了提升，那么以前觉得棘手的问题，此时也许很容易就能解决了。

定期回顾案例是很好的做法，有助于提升自己的能力。老白每次写书时，都会回过头去翻看大量以前做过的案例，从中挑选一些作为写作的素材。这种回顾对老白来说是受益无穷的。在翻看时，老白会发现一些自己十分满意的案例，其中包含很多值得探讨的内容，而另外一些案例只

是表面上解决了问题，但其实当时并没能抓住问题的实质，解决问题仅仅是运气而已。

另外，我们还应该创建自己的问题处理档案，并建立闭环管理制度，给自己制定一个问题关闭的期限，这样有助于深入地分析、总结问题。同时，定期回顾机制也可以进一步提升自己的能力，并提供更多的学习途径。因此老白建议大家，无论能力如何，都应尽快建立一套适合自己的案例管理机制。

15.7　灵活运用你的知识

Oracle RDBMS 是十分复杂的系统，我们不能只依靠简单的原理来处理 RDBMS 的系统问题。很多 DBA 都热衷于掌握一些特别简单实用的原则，比如 DB Cache 的命中率不能低于 90%，日志切换的时间不能小于 15 分钟等。但实际上，在掌握这些原则的同时，也陷入了一个罗生门。尽管这些知识对于分析数据库是有帮助的，我们也可以通过这些知识对数据库制定定量的分析准则，但如果不能灵活应用，就可能会带来很多不利的影响。

在简单的环境中，某些原则可能很容易发挥作用，比如，我们不考虑其他的因素，单从日志切换对系统造成的影响来看，过于频繁的日志切换肯定对系统不利，是应该尽可能避免的。但是有些情况下，这个原则可能会起反作用。

前些年，我们给一个客户做巡检，每三个月一次。我第一次到现场时就发现这个系统的日志量很大，几乎 20 多秒就会切换一个日志，而日志文件的大小是 100 MB。这种设置明显是很不合理的，但我并没有马上下结论，而是分析了日志切换可能给系统带来多大的影响。经过分析发现，LOG FILE SWITCH 事件的等待时间很短，而且占整个系统的等待比重并不高，只有不到 1%，这就说明目前的日志切换频率并没有对系统产生较大的影响。另外，CF 锁所占的等待比重也不高，说明日志切换并没有对 CF 锁造成太大的影响。目前这个系统存在一个本地 DATAGUARD、一个远程 DATAGUARD（通过日志以手工压缩传输方式完成）和一个逻辑 DATAGUARD，从这种配置来看，较小的日志文件还是对这些 DATAGUARD 有些帮助的。于是我建议客户维持现状，暂时不要扩大日志文件。

一年后，这个系统的 REDO LOG 量持续增长，日志切换的间隔进一步缩小，CF 锁等待和 LOG FILE SWITCH 等待的平均等待时间也在不断增长。在这种情况下，生产库因为日志文件过小而出现故障的概率也在不断增长。尽管 DATAGUARD 仍然存在，但是我们必须加大 REDO LOG 文件，从而确保生产库的正常运行。虽然基础环境并没有变化，但这时对 REDO LOG 文件做出调整和一年前保持文件大小不变，同样都是正确的。

水无定势，大道无形。优化也是一样，实施者需要具备广博的知识，知识面越广，越能够在纷繁的头绪中选取最佳的技术路线。也许这样说对很多 DBA 要求太高了，但是经验和知识需要积累，不是看几本书就可以实现的。确实，我最近在写 DBA 日记时翻阅了很多几年前的文档，发现以前在处理很多案例的时候，都犯了不少的错误，起码处理得不够圆满，可能做这一行久了，看东西就变得苛刻起来了。

一个人不可能是全才，不可能无所不知，犯错误是难免的，但掌握的知识多一些，犯错误的

机率就会变小。去年给一个客户做优化，当时看到系统都使用了绑定变量，所以觉得大不了就用存储概要（stored outlines）来解决执行计划的问题。但到了最后，问题真的来了，优化后绝大多数模块的性能都提高了数倍，只有一个模块性能下降了50%多。经过检查发现，是一张表和一个视图做连接，优化器无法主动进行视图合并，导致这个模块性能下降。后来经过手工测试，添加合并提示后性能问题就解决了。但当时我没有想到这个提示居然无法通过存储概要合并。无奈之下，我只能将相关数据压缩并发送，希望印度朋友能帮忙解决，但最终也没能得到回应，于是自己找来一份文档，才了解到合并是无法通过存储概要添加的。我通过这个项目长了学问，但也被折腾得够呛。最后经过一番软磨硬泡，客户才接受了那个模块性能必须下降的结果，至此，终于了结了这个项目。我想今后再去实施类似的项目时，就会多一个心眼了。

有一次，和几个网友讨论存储概要的问题，一位朋友问道，为什么不使用SQL PROFILE呢，这对优化SQL不是更有效？确实在绝大多数情况下，SQL PROFILE更为有效，但真的碰到一些不太可能发生的少数情况时，存储概要可能会比SQL PROFILE更适合。优化无定式，合适就好。

算起来，从第一次给客户优化系统到现在已经10多年了。在这些年里，接触过不同的客户和系统，对于优化工作的体会也越来越深刻。刚开始做优化时，总是希望找出系统中所有存在问题的地方，然后逐个进行调整。由于对Oracle的基本原理认识不够，并且对优化的认识也仅限于调整不合理部分的浅层次上，因此经常会遇到一些事与愿违的情况。实际上，进行优化工作时需要灵活运用知识。首先，优化是基于目标的，我们的最终目的是达到这个目标，而不是做优化。其次，目标的合理性决定了优化项目的成败。刚开始从事优化工作时，我会将所有能够调整的问题一次性处理完毕，即使有些调整给系统性能带来的好处只有不到0.1%。生产系统的不确定因素很多，而一些参数方面的调整本身就是双刃剑，如果无法预期其带来的影响，那么这种调整就是存在风险的，在实施的时候就应该慎重考虑。现在我再做优化项目时，往往会根据用户的优化目标进行分析，并在此基础上制定方案，实施的结果一般都会超出客户的期望，但是我不会在生产系统上进行一些没把握的调整。锦上添花的事情有时候是需要慎重考虑的，弄不好就会变成画蛇添足。这种情况下，需要针对具体问题进行分析，不应拘泥于某个知识点，而应将所掌握的知识灵活运用到实际的优化工作中。比如，我们经常被教导，SQL一定要使用绑定变量，这句话在绝大多数场合是对的，但在某些场合却不一定正确。在10*g*数据库中，如果某个字段是倾斜的，由于柱状图的存在，可能不使用绑定变量会更好一些。可能有些朋友会说，在11*g*版本中，随着ACS技术的出现，是不是就可以放心地使用绑定变量了呢？是的，在绝大大数情况下确实如此，不过老白也碰到过一种情况，尽管ACS已经发挥了作用，但在一个游标下还是产生了大量不可共享的子游标，这导致CURSOR:PIN S WAIT X等待十分严重。最后取消了绑定变量，才解决了这个问题。

另外需要注意的一点是，“1+1”其实并不一定大于1。在处理问题的过程中，应抓住主要矛盾，解决主要问题，而不是胡子眉毛一把抓。很多调整之间都存在关联性，甚至是互斥的，不合理的调整可能带来更糟的结果。这也需要我们灵活运用所学的知识，而不能墨守成规。在大多数情况下，如果CPU使用率接近100%，那么加大DB Cache就很可能会导致更严重的CPU争用，但是这也不是绝对的。老白就碰到过一个类似的案例，通过分析发现，由于某个查询相关的一张

表的数据存放在 I/O 能力较差的文件上，而此时 I/O 的整体性能也不佳，平均单块读操作响应时间达到 11 毫秒，这使得这条 SQL 的执行时间从 0.01 秒上升为 0.1 秒，从而导致了大量的查询积压，提高了 CPU 的使用率。通过加大 DB Cache，提高了 I/O 的整体响应时间，这条 SQL 的平均执行时间从 0.1 秒下降为 0.04 秒，查询积压消除了，CPU 使用率恢复到 80%左右。

不要相信什么规则，实际上，并没有条条框框的限制，任何实现目标的方法都是可以使用的。对于一个初级 DBA 来说，老 DBA 可能会告诉你，什么是对的，什么是不对的。而对错原本就是相对的，如果已经深刻地理解了 Oracle 以及系统优化的原理，那么就可以像金庸小说里的内功高手一样，无招胜有招，进入到更高层次的自由发挥境界。

15.8　DBA 需要与时俱进

Oracle 的数据库技术发展很快，这一点对于每个 DBA 来说，都是既痛苦又幸福的事情。最近这 10 多年，Oracle 公司基本上以每 5 年一个新版本的速度进行更新。记得 2004 年在做一个优化项目时，还和几个搭档讨论过 Oracle 发展的问题，其中一个人说他已经从 7.0 干到 8*i* 了，8*i* 也许就是他学习的最后一个版本了，绝对不会再学 9*i* 了。不过没多久，我们又接了一个 9*i* 的优化项目，于是我看到他房间里又多了几本 9*i* 的书籍。这回他再次向我发誓，9*i* 将是他职业生涯中学习的最后一个版本，坚决不接触 10*g*。不过这个誓言后来又变成了 11*g*，这回他没有再次食言，因为他彻底改行了，去一家电商网站做起了 CIO。

每 5 年更新一个版本，这对 DBA 提出了很大的挑战。Oracle 技术发展得如此之快，这意味着已经约定俗成的一些概念和技巧很快就会过时。如果 DBA 总是以老观念来面对新的数据库，那么肯定会有问题的。我经常在网络上和一些朋友进行沟通，发现他们经常会提到的一些网络上的文章，而文章中提出的一些观点，以及用到的一些技术，都是很早以前 8.0 时代的技术，甚至有些是基于 7.0 数据库的。有些知识已经严重过时了，甚至有些观点放到现在的 10*g*、11*g* 环境中已经是错误的了。

在 10 多年前，那时候的硬件资源十分昂贵，32 位的系统是主流系统，因此系统配置往往很低。对于 DBA 来说，优化的重点是如何协调使用这些系统资源，使之能够达到一个比较优化的平衡点，因此各种辗转腾挪的小技巧被使用到了极致。最典型的示例就是基于共享池的优化操作，那时的 DBA 必须熟练掌握共享池的优化技术。哪怕有几个参数没有设置妥当，都可能导致十分严重的性能灾难，甚至出现宕机的故障。

那时候数据库的主流版本是 8.0 和 8*i*，大多数服务器只配备了 4 ~ 8 GB 的物理内存和 2 ~ 4 个 CPU。一般来说，共享池也只有 100 MB ~ 1 GB，500 MB 的共享池配置已经算是比较大的了。由于共享池容量较小，因此，一旦共享池出现争用，将会导致十分严重的后果。

```
Cache Sizes
~~~~~~~~~~~
db_block_buffers:     786432          log_buffer:             3145728
db_block_size:          8192          shared_pool_size:     681574400
```

上面是一个典型的 8*i* 数据库的 CACHE 配置，这套系统是一个电信的计费系统，配置了 650 MB

的共享池和 6144 MB 的 DB Cache，这在当年也算是非常大的系统了。对于这样的系统，如果每秒存在几百个 SQL 解析，那么共享池就会受到很大的冲击，一旦某些配置不合理，或者业务高峰时对某些存储过程、视图进行了编译，就可能导致共享池出现严重的性能故障。因此在 8*i* 时代，共享池中的参数设置需要十分精细，而一般的优化项目也会将共享池的优化工作作为重点。在这种情况下，使用绑定变量、调整 CURSOR_SHARING、OPEN_CURSORS、SESSION_CACHED_CURSORS 等参数就会变得十分关键。记得 10 多年前我曾优化过一套 8*i* 的系统，多数情况下系统运行正常，但有时会变得非常慢，甚至会出现几乎无法工作必须强制重启的情况。而且这种情况没有什么规律，在业务高峰的月底、月初都没问题，而业务最闲的月中反而经常出问题。客户找了很多公司帮忙，甚至原厂都去了很多次，但一直没能解决问题。当我找到问题的原因向他们汇报时，大家都感到有些啼笑皆非，原来是系统中的一个应用采用了短连接方式，经常会由于大量的 LOGON/LOGOFF 操作导致共享池出现问题。之前，Oracle 公司的工程师曾建议对这部分应用使用共享服务器模式，自从使用了该模式后，系统变得稳定了。这是一个不错的优化操作，使用共享服务器模式减少了进程启动和关闭的开销，使得系统更为稳定，但随着这个应用业务量的增长，共享服务器模式的会话对大型池（large pool）的需求也在不断增长，当大型池不足时,问题就出现了。这部分应用变慢会导致其对共享池中其他对象的 PIN 时间变长，从而产生连锁反应，导致共享池争用越来越严重，并最终出现问题。找到了问题的原因，解决起来就很容易了，只需将大型池从 100 MB 调整到 200 MB，这个问题就解决了。当时我还特意嘱咐客户，由于目前内存有限，无法将大型池设置得太大，如果业务量还在不断扩大，那么过一段时间共享池可能还会不足，一定要注意监控。

这个问题如果放到现在就很好解决了，在物理内存十分廉价的今天，一台微机服务器往往都可以配置几十 GB 甚至几百 GB 的物理内存，随便给共享池分配 500 MB 的内存，就万事大吉了。随着内存价格的不断下降，拥有大内存、大量 CPU 的服务器已经不再是奢望了。在近些年的优化项目中，我经常通过设置足够大的共享池来解决一些问题，甚至有些问题都没有进行特别深入的分析，只分析了其浅层的问题原因，而这些调整往往都是立竿见影的。起初客户还有些担心，特别是在 8*i* 时代，一般的共享池配置都是几百 MB，而在那个时候，我就经常在一些 64 位的系统上配置大共享池。

有一次，我在做一个优化项目时，建议客户将共享池从 512 MB 调整到 2 GB。客户本身也是一名资深的 DBA，他对共享池的结构非常清楚，对这项调整感到十分担忧。8*i* 的 FREELIST 只有 10 多个 BUCKET，那么大的内存挂在几个列表上，会不会导致 FREELIST 争用，从而影响性能呢？其实这也是当时很多 DBA 所担心的，甚至有些观点认为共享池设置过大会严重影响系统的性能。确实，将共享池设置得过大，是有可能引起严重的性能问题，这种案例在 10 多年前也比比皆是。不过这些故障都是有前提的，那是在基于很高的并发量、较小的系统资源，特别是 CPU 已经出现了瓶颈的情况下发生的。而我当时面对的这个案例不同，虽然客户使用的还是 8*i* 版本，但是系统的物理内存有 48 GB，CPU 也有 24 个，CPU 资源是十分充足的，在这种情况下，较大的共享池的作用是正面的，其负面影响几乎可以忽略。事实证明，调整了共享池后，系统变得十分稳定，这次优化是成功的。

在这个案例中，之前的好几个 DBA 都曾经尝试对系统进行优化，也都发现了共享池不足给系统带来了很大的影响，都在试图减少应用对共享池的使用。不过在应用架构不能够修改的情况下，这些努力收效甚微。他们也曾经试图加大共享池，但是并不敢将共享池设置得过大，曾经有一个 DBA 将这个系统的共享池从 512 MB 加大到 800 MB，但没有收到明显的效果，于是又回退了。他们为什么会失败，而老白却能够成功呢？这是因为他们并没有综合分析问题的根源，而是被以往的案例和大家普遍认可的“相对真理”所束缚了。因此在调整共享池这个问题上谨小慎微，不敢大胆突破。老白并没有被那些所谓的专家观点、历史教训所束缚，而是从本质上分析了以往设置较大共享池导致性能下降的案例，其中大多数都是由于 CPU 资源存在瓶颈，共享池变大后，并发量的增加导致了共享池闩锁争用加剧，从而使得 CPU 资源消耗更大，并最终导致了性能问题。基于这样的分析，老白认为在 CPU 资源能够得到保证的前提下，使用较大的共享池是可行的，因此才能够大胆地将共享池从 512 MB 加大到 2 GB。

从第一个大型优化项目开始算起，老白从事 Oracle 性能优化工作也差不多有 10 年的时间了，在这 10 年中，优化的重点和方法在不断地变化。现在回过头来看 10 年前的一些项目资料，我发现很多优化技术都已经有些陌生了。在 10 多年前，回滚段的优化是十分重要的，特别是在电信计费这类变更十分频繁的系统中。在 7.0、8.0 和 8*i* 时代，大型计费系统的优化都离不开回滚段的优化。设置合适的回滚段数量以及 OPTIMAL_SIZE 参数值，都是十分重要的，甚至每个扩展的大小都可能成为系统故障的元凶。那个年代的 DBA，都掌握一个技巧，就是必须在每个系统中留出几个较大的回滚段，作为处理大事务的专用回滚段。一旦要对计费数据进行较大的更新操作，就必须指定使用这些大回滚段。否则就可能出现 SQL 执行了 10 多个小时后，由于回滚段不足而失败。

即便如此小心地优化回滚段，ORA-1555 错误仍然是一个十分考验 DBA 技巧的问题。记得那时 DBA 面试的时候，如果某个应聘人员能够很圆满地回答如何解决 ORA-1555 的问题，那么他被录用的机会就很大了。有不少 DBA 靠着这一招，就可以“混迹江湖”了。

不过随着 9*i* 版本的出现，这些技术都变得无用了。这让很多老 DBA 感到十分沮丧，“现在的年轻人只会设置 UNDO_RETENTION 参数，连专用回滚段都不知道”，这种观点其实只是酸葡萄心理而已。既然有了 UNDO 自动管理，为什么还要回过头去使用复杂的手工管理模式呢？虽然在某些极端情况下，UNDO 手工管理还有一定的作用，但在一般情况下，我们可以放心大胆地去使用 UNDO 自动管理。

在以前的一个优化项目中，客户的 DBA 曾问我，现在默认的 UNDO 空间只有 40 GB，如果不够用怎么办。我告诉他，那就给它开 200 GB。10 多年前我们在小心谨慎地优化回滚段时，经常为了能腾出 1～2 GB 的空间给某个 RBS 而煞费苦心，而现在一块大点的硬盘都有 2 TB 的容量了，几百 GB 的存储空间简直是手到擒来的事情。在这种情况下，我们有什么理由不给 UNDO 多分配一些空间呢？

如果 DBA 不能与时俱进，还守在一些陈芝麻烂谷子里面过日子，那么可以预见，他们必然会付出大量的劳动，而只能收获很小的成果。Oracle 这些年在技术上的进步是巨大的，确实如拉里所说，Oracle 10*g* 是一个充分自管理的数据库，DBA 应该充分利用这些新特性，将自己从一些

繁杂的事情中解脱出来，去考虑更需要投入精力的问题。

老白曾经不止一次说过，使用 10*g* 数据库而不用 EM 管理器是极大的浪费。确是如此，这些年 10*g* 版本的应用已经十分普及了，但事实上 2011 年就已经进入了 10*g* 的延长维保期，Oracle 会很快停止对 10*g* 版本的支持，可能大规模迁移到 11*g* 就是最近一两年的事情了。而绝大多数的 DBA 还在用 9*i* 的方式管理着 10*g* 的数据库，还抱着一堆脚本不放，用陈旧落后的手段维护着自己的系统。10*g* 数据库中一个十分耀眼的新特性就是其新的管理基础架构，这个基础架构是构建自管理数据库的核心，而 EM 管理器和该基础架构的集成十分紧密，通过 EM 管理器来使用 10*g* 版本的新管理特性是最为简便的方法。

11*g* 版本在这方面走得更远，它是一个充分仪表化的数据库，我们可以在 Oracle 的管理框架中更为便捷地管理数据库。这对于还没有习惯使用图形界面的 DBA 来说，是一个更大的挑战。

在本节的最后，老白再一次呼吁广大 DBA，现在是摒弃固有的成见，与时俱进、勇敢接触新技术的时候了；现在是放下手头的脚本，看着图形化仪表来管理数据库的时候了。

15.9 多表连接的优化技巧

如果一个系统突然变慢，并且 AWR 报告中某条 SQL 的 BUFFER GET 很高，那么我们就需要使用 AWRSQRPT 脚本生成一个 SQL 报告。此时我们会发现，出现问题的是一条十分复杂的 SQL，这条 SQL 包含多层嵌套，其执行计划有几十行甚至上百行之多。遇到这样的 SQL 我们是否就束手无策了呢？实际上，任何复杂的 SQL 最终都可以分解为多次二表连接，因此完全可以从这里找到入手点，从而解决这个问题。比如，我们在一次优化中发现某条 SQL 对系统的影响很大，于是通过 AWRSQRPT 脚本获取了完整的 SQL 文本，如代码清单 15-1 所示。

代码清单 15-1

```
select acct_id,billing_cycle,bill_item_name,item_source_name,to_
char(sum(amount),'FM999999990.90') amount,stateName,stateDate
from( select a.acct_id ,c.billing_cycle,d.bill_item_name,e.name
item_source_name,a.amount,f.name stateName,to_char(a.state_date,'
yyyy-mm-dd hh24:mi:ss') stateDate  from  (select acct_item_type_
id ,billing_cycle_id,acct_id,item_source_id,state,state_date,sum
(amount/100) amount  from( select * from  acct_item  where acct_
id =165440341 and serv_id =169002286374  and billing_cycle_id in
(10906) ) group by acct_item_type_id,billing_cycle_id,acct_id,
item_source_id,state ,state_date) a, (select billing_cycle_id ,to_
char(cycle_end_date -1,'yyyy-mm') billing_cycle from billing_cyc
le where state in('10A','10R','10E','10D')  and billing_cycle_id
in(10906))c ,(select  x.acct_item_type_id ,y.bill_item_name,x.
item_source_id  from bill_item_acct_item x ,bill_item y , (select
distinct invoice_require_id from serv_acct where acct_id=165440
341 and serv_id =169002286374 ) ee , bill_requement ff,bill_form
at_bill_item gg where x.bill_item_type_id = y.bill_item_type_id
and ee.invoice_require_id = ff.require_id  and y.classify='55A'
and ff.bill_format_id = gg.bill_format_id and gg.bill_item_type
_id = x.bill_item_type_id)d ,acct_item_source e, (select domain,
name from v_domain where table_name ='ACCT_ITEM' and field_name=
```

```
'STATE' ) f  where a.billing_cycle_id = c.billing_cycle_id and a
.acct_item_type_id= d.acct_item_type_id and a.item_source_id = d.
item_source_id and a.item_source_id =e.item_source_id and e.ite
m_source_type='52A' and a.state=f.domain ) group by acct_id  ,
billing_cycle,bill_item_name,item_source_name,stateName,stateDate
order by billing_cycle
```

想看懂这条 SQL 几乎是不可能的,这种 SQL 简直会让我们崩溃。其实,对于经验丰富的 DBA 来说，问题并没有那么严重。首先，我们需要对这条 SQL 进行格式化，很多工具都可以完成此工作（老白在使用工具之前一直是手工来完成格式化操作的，整理这样一条 SQL，可能需要几个小时），老白最喜欢使用的是 Oracle 的 SQL DEV 工具。我们可以从格式化后的 SQL 中看出层次关系，这对于下一步优化工作至关重要。因此，如果没有合适的 SQL 格式化工具，就无法进行后续工作。代码清单 15-2 是格式化后的 SQL 文本。

代码清单 15-2

```
SELECT acct_id,billing_cycle,bill_item_name,item_source_name,
  to_ CHAR(SUM(amount),'FM999999990.90') amount,  stateName,  stateDate
FROM
  (SELECT a.acct_id , c.billing_cycle,d.bill_item_name,  Item_source_name,
    a.amount,    f.name stateName, TO_CHAR(a.state_date,'yyyy-mm-dd hh24:mi:ss')
stateDate
  FROM
    (SELECT acct_item_type_ id ,     billing_cycle_id, acct_id, item_source_id,
      state,   state_date,  SUM (amount/100) amount
    FROM
      (SELECT *
      FROM acct_item
      WHERE acct_ id =165440341
      AND serv_id =169002286374
      AND billing_cycle_id IN (10906)
      )
    GROUP BY acct_item_type_id, billing_cycle_id, acct_id,item_source_id,state ,state_date
    ) a,
    (SELECT billing_cycle_id ,  to_ CHAR(cycle_end_date -1,'yyyy-mm') billing_cycle
    FROM billing_cycle
    WHERE state          IN('10A','10R','10E','10D')
    AND billing_cycle_id IN(10906)
    )c ,
    (SELECT x.acct_item_type_id , y.bill_item_name, x.i tem_source_id
    FROM bill_item_acct_item x ,   bill_item y ,
      (SELECT DISTINCT invoice_require_id
      FROM serv_acct
      WHERE acct_id=165440341
      AND serv_id=169002286374
      ) ee ,
      bill_requement ff,  bill_form at_bill_item gg
    WHERE x.bill_item_type_id = y.bill_item_type_id
    AND ee.invoice_require_id = ff.require_id
    AND y.classify            ='55A'
    AND ff.bill_format_id     = gg.bill_format_id
    AND gg.bill_item_type _id = x.bill_item_type_id
    )d ,
    acct_item_source e,
```

```
        (SELECT domain,    name
        FROM v_domain
        WHERE table_name ='ACCT_ITEM'
        AND field_name   = 'STATE'
        ) f
      WHERE a.billing_cycle_id = c.billing_cycle_id
      AND a .acct_item_type_id = d.acct_item_type_id
      AND a.item_source_id     = d .item_source_id
      AND a.item_source_id     =e.item_source_id
      AND e.ite m_source_type  ='52A'
      AND a.state              =f.domain
      )
    GROUP BY acct_id , bi lling_cycle, bill_item_name,  item_source_name,stateName,
    stateDate
    ORDER BY billing_cycle
```

在上述 SQL 中存在大量的 inline 视图，这使得整个结构十分复杂。我们需要进行相应的简化，找到这条 SQL 中对性能影响最大的部分，由于该 SQL 带有子查询，因此可以先将子查询单独拿出来分析，如代码清单 15-3 所示。

代码清单 15-3

```
     (SELECT a.acct_id , c.billing_cycle,d.bill_item_name,  Item_source_name,
        a.amount,    f.name stateName, TO_CHAR(a.state_date,'yyyy-mm-dd hh24:mi:ss')
    stateDate
      FROM
        (SELECT acct_item_type_ id ,     billing_cycle_id, acct_id, item_source_id,
          state,   state_date,  SUM (amount/100) amount
        FROM
          (SELECT *
          FROM acct_item
          WHERE acct_ id        =165440341
          AND serv_id           =169002286374
          AND billing_cycle_id IN (10906)
          )
        GROUP BY acct_item_type_id, billing_cycle_id,
    acct_id,item_source_id,state ,state_date
        ) a,
        (SELECT billing_cycle_id ,  to_ CHAR(cycle_end_date -1,'yyyy-mm') billing_cycle
        FROM billing_cycle
        WHERE state          IN('10A','10R','10E','10D')
        AND billing_cycle_id IN(10906)
        )c ,
        (SELECT x.acct_item_type_id , y.bill_item_name, x.i tem_source_id
        FROM bill_item_acct_item x ,   bill_item y ,
          (SELECT DISTINCT invoice_require_id
          FROM serv_acct
          WHERE acct_id=165440341
          AND serv_id  =169002286374
          ) ee ,
          bill_requement ff,  bill_form at_bill_item gg
        WHERE x.bill_item_type_id = y.bill_item_type_id
        AND ee.invoice_require_id = ff.require_id
        AND y.classify            ='55A'
```

```
    AND ff.bill_format_id     = gg.bill_format_id
    AND gg.bill_item_type _id = x.bill_item_type_id
    )d ,
    acct_item_source e,
    (SELECT domain,    name
    FROM v_domain
    WHERE table_name ='ACCT_ITEM'
    AND field_name   = 'STATE'
    ) f
  WHERE a.billing_cycle_id = c.billing_cycle_id
  AND a .acct_item_type_id = d.acct_item_type_id
  AND a.item_source_id     = d .item_source_id
  AND a.item_source_id     =e.item_source_id
  AND e.ite m_source_type  ='52A'
  AND a.state              =f.domain
  )
```

这部分语句仍然嵌套了子查询，我们可以先从 FROM 语句开始向下分析，可以看出 FROM 语句后面存在几张表和几个 inline 视图，其别名分别为 a、c、d、e、f，如代码清单 15-4 所示。

代码清单 15-4

```
视图 1:
   (select
acct_item_type_id ,billing_cycle_id,acct_id,item_source_id,state,state_date,sum
(amount/100)
    amount from ( select * from  acct_item  where acct_id =165440341 and serv_id
=169002286374
    and billing_cycle_id in (10906)) group by
acct_item_type_id,billing_cycle_id,acct_id,
    item_source_id,state ,state_date
   ) a,
视图 2:
   (select billing_cycle_id ,to_char(cycle_end_date -1,'yyyy-mm') billing_cycle
from billing_cycle
    where state in('10A','10R','10E','10D') and billing_cycle_id in(10906)
    )c
视图 3:
  (select  x.acct_item_type_id ,y.bill_item_name,x.item_source_id from
bill_item_acct_item x ,bill_item y ,
    (select distinct invoice_require_id from serv_acct where acct_id=165440 341 and
serv_id =169002286374
    ) ee ,bill_requement ff,bill_form at_bill_item gg
     where x.bill_item_type_id = y.bill_item_type_id and ee.invoice_require_id =
ff.require_id  and
    y.classify='55A'  and ff.bill_format_id = gg.bill_format_id and
gg.bill_item_type_id = x.bill_item_type_id
     )d
表 1:
acct_item_source e,
视图 4:
(select domain,name from v_domain where table_name ='ACCT_ITEM' and
field_name='STATE'  ) f
```

经过上面的分析，可以看出这个查询由 5 个主要部分关联而成，实际上，我们需要判断的是，针对这 5 个表/视图，应该采取哪种连接顺序。确定连接顺序需要从连接关联条件和过滤条件开始。关联条件如代码清单 15-5 所示。

代码清单 15-5

```
a.billing_cycle_id = c.billing_cycle_id and
   a.acct_item_type_id= d.acct_item_type_id and
   a.item_source_id = d.item_source_id and
   a.item_source_id =e.item_source_id and
   e.item_source_type='52A' and
   a.state=f.domain
```

从这里可以看出，a 表是核心，其他表均通过 a 表来关联。这个查询的过滤条件很少，只有 e 表上存在一个过滤条件。通过对这些 inline 视图的分析，我们并没有发现其中存在重叠部分，因此不需要进行深入的视图合并。这里，我们首先考虑 a+e 的连接组合，其中，a 是一个视图，e 是一张表。将 a 和 e 的过滤条件及连接条件代入，就得到了一个 a+e 连接的等价 SQL，如代码清单 15-6 所示。

代码清单 15-6

```
Select  a.acct_item_type_id , a.billing_cycle_id,acct_id, a.item_source_id,a.state,
a.state_date,sum(a.amount/100)amount from
  (select
acct_item_type_id ,billing_cycle_id,acct_id,item_source_id,state,state_date,sum
(amount/100)
    Amount from ( select * from  acct_item  where acct_id =165440341 and serv_id
=169002286374
    and billing_cycle_id in (10906) ) group by
acct_item_type_id,billing_cycle_id,acct_id,
    item_source_id,state ,state_date
   ) a, Acct_item_source e
Where
 a.item_source_id =e.item_source_id and
 e.item_source_type='52A'
```

接下来，需要分析这个等价 SQL 的执行计划，我们已经将分析集中在一个二表连接的优化上了。因此，分析相对就简单了许多。

从上面的 SQL 可以看出，a 的数据来自于账目表 ACCT_ITEM（这是电信账务系统中一张很重要的表，存放着所有的账目），而过滤是通过 acct_id、serv_id（服务 ID）和 billing_cycle_id（账期）这三个等于条件进行的，这样过滤出来的数据量不会很大，一般只有几条到几百条。

下一步我们可以将这个子查询产生的数据存放在一张临时表中，然后判断 a+e 关联的结果集以及其他几个表连接时产生的结果集的数量，从而判断下一步和哪张表关联会比较高效（如果分析能力足够强，或者对业务比较了解，也可以直接通过业务关系进行分析，这个示例只是演示在不了解整个业务的情况下该如何分析 SQL），如代码清单 15-7 所示。

代码清单 15-7

```
CREATE TABLE TMP_TBL1 AS
Select  a.acct_item_type_id , a.billing_cycle_id,acct_id, a.item_source_id,a.state,
a.state_date,sum(a.amount/100) amount from
  (select
acct_item_type_id ,billing_cycle_id,acct_id,item_source_id,state,state_date,sum
(amount/100)
    amount from ( select * from  acct_item  where acct_id =165440341 and serv_id
=169002286374
    and billing_cycle_id in (10906)) group by
acct_item_type_id,billing_cycle_id,acct_id,
      item_source_id,state ,state_date
   ) a, Acct_item_source e
Where
 a.item_source_id =e.item_source_id and
 e.item_source_type='52A' ;
EXEC DBMS_STATS.GATHER_TABLE_STATS(OWNNAME=>'...',TABNAME=>’TMP_TBL1);
```

接下来，我们可以改写 SQL，评估其他表和 TMP_TBL1 连接时的执行计划，如代码清单 15-8 所示。

代码清单 15-8

```
select c.*
(select billing_cycle_id ,to_char(cycle_end_date -1,'yyyy-mm') billing_cycle
   from billing_cycle
   where state in('10A','10R','10E','10D')  and
   billing_cycle_id in(10906)
  )c ,tmp_tbl1 t
where
 t.billing_cycle_id = c.billing_cycle_id
```

如果这个查询非常复杂，可以按照前面的方式进行分拆，直到能够很清晰地进行分析为止。这样，经过几轮的分拆和分析，就可以将一个复杂的 SQL 执行计划分析得很透彻了。

尽管目前的 SQL 分析工具功能很强大，已经可以分析大多数 SQL 的性能了，但针对特别复杂的 SQL，工具的能力毕竟还是有限的，而我们又经常会遇到这类复杂的 SQL，因此，掌握好本节介绍的方法，才是处理复杂 SQL 的关键。

15.10　理论如何联系实践

老白在本书中多次提到理论和实践的结合。如果只有理论而无实践，理论只是空中楼阁，中看不中用；但如果只有实践而无理论，又容易流于表面，经常是治标不治本。理论联系实际这句话大多数 DBA 都清楚，但是谈到如何在实际工作中实现，恐怕很多人就十分迷茫了。

要做到理论联系实际，首先需要比较精准地掌握理论知识，但这并不是要求大家都去深入研究某个操作的 Oracle 内部实现算法，其实只要理解了大体的概念，基本上就够用了。由于精力有限，一个人不可能把成千上万人开发出来的、如此庞大的软件系统了解得十分透彻，不过我们可

以通过 Oracle 公布的一些官方资料以及网络上类似 DSI 这样的资料，粗略地了解 Oracle 各个功能模块、组件的基本原理和算法。大多数 DBA 学习理论知识的渠道并不是 *Oracle Concepts* 这本书，而是在网络上拼命搜罗一些高深的底层核心知识，这样掌握的知识点十分零散，有些人能够将部分知识点串在一起，而大多数人了解的只是一个个孤立的点，因此这些知识很难发挥作用。

前段时间，在一个优化项目中，我们发现系统的共享池问题很大，相关分析报告如下：

```
Cache Sizes
~~~~~~~~~~~~~~~~~                  Begin      End
              Buffer Cache:      4,288M     4,288M      Std Block Size:        16K
          Shared Pool Size:      3,856M     3,856M          Log Buffer:    14,292K

Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                 Redo size:              32,572.79              6,444.27
             Logical reads:              20,006.94              3,958.21
             Block changes:                 206.09                 40.77
            Physical reads:                  79.91                 15.81
           Physical writes:                   4.73                  0.94
                User calls:                 163.69                 32.38
                    Parses:                 122.99                 24.33
               Hard parses:                  16.93                  3.35
                     Sorts:                  27.48                  5.44
                    Logons:                   0.10                  0.02
                  Executes:                 131.80                 26.07
              Transactions:                   5.05

 % Blocks changed per Read:     1.03    Recursive Call %:     80.29
Rollback per transaction %:    22.79       Rows per Sort:    459.53

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   100.00       Redo NoWait %:   100.00
               Buffer Hit %:    99.60    In-memory Sort %:   100.00
              Library Hit %:    93.36        Soft Parse %:    86.23
         Execute to Parse %:     6.68         Latch Hit %:    99.91
Parse CPU to Parse Elapsd %:     8.40     % Non-Parse CPU:    94.11

 Shared Pool Statistics        Begin    End
                               ------  ------
             Memory Usage %:   87.98    85.28
    % SQL with executions>1:   83.42    67.19
  % Memory for SQL w/exec>1:   91.20    80.17

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                          % Total
Event                      Waits  Time(s)   Avg Wait(ms)   Call Time    Wait Class
------------------------- ------ --------- -------------- ------------ -----------
CPU time                            2,547                          44.4
latch: library cache       5,813    1,593            274           27.8  Concurrency
latch: shared pool         4,932    1,303            264           22.7  Concurrency
latch free                   721      189            262            3.3        Other
db file sequential read   31,799      139              4            2.4     User I/O
---------------------------------------------------------------------------------
```

大家可能马上就能发现库缓存和共享池闩锁的等待十分严重，系统总体等待时间的占比超过了 50%，而且平均每次等待的时间在 260 毫秒左右，而从后面的等待事件明细来看，平均每个事务等待这些闩锁的次数为 0.32，也就是说，在一个事务中，仅仅库缓存和共享闩锁的等待时间就超过了 160 毫秒。而且这个系统已经多次出现由“行缓存对象等待时间过长”导致的挂起和宕机事件。我们分析了这个应用系统的特点，由于该系统中的很多统计查询结果都是先存储在临时表中，然后再将结果显示给客户，因此 SQL 的重用率较低，硬解析的比重很高，这样就导致了共享池的命中率不高，仅为 93%，而软解析的比例仅为 86%。按理说这样的系统，如果使用了共享内存自动管理，那么共享池应该会被调整得很大。但从报告中看到，共享池仅为 3 GB 多，于是我们检查了 SGA 的配置情况，发现 SGA_TARGET 参数设置为 28 GB，共享池配置了 3 GB，DB CACHE 配置了 4 GB，保留池配置了 4 GB，回收池配置了 16 GB。

从这些配置信息来看，客户对这个系统做了精心的设计，而且设置这些参数的 DBA 肯定对 SGA 的基本原理比较了解，但回收池的配置着实让老白十分不解。通过沟通了解到，设置如此大的回收池的目的有两个：一是针对几张很大的表（所有表都超过 1 GB，最大的表有 10 GB 多），其数据使用十分频繁，而且经常进行分区扫描，为了确保这些表的高效访问，才将其放入回收池，之所以不将它们放入保留池是为了防止其冲击池中的数据；二是为了防止共享池过度扩张。

从第一个目的来看，这种设计十分巧妙，不仅通过较大的回收池实现了类似保留池的功能，而且不会因为某个较大的扫描操作影响保留池的性能，这种使用方法虽然不合常规，但也是没有任何问题的。不过第二点老白就觉得不可思议了，为什么要以回收池占用所有的内存来防止共享池扩张呢？

经过了解，原来是由于系统中大量临时表的使用，共享池的争用问题一直十分严重，对系统性能和稳定性都造成了很大的影响。他们在网络上查阅了大量的资料，其中有不少文章指出，共享池过大会增加闩锁争用，从而导致更为严重的共享池性能问题。实际上，“过大”这个概念本身就十分模糊，于是他们进行了一系列的实验，发现将共享池配置为 3 GB 时性能最佳，超过 4 GB 性能反而有所下降。根据此实验结果，他们最终设计了这样一套 SGA 配置方案。

经过综合分析，老白的优化建议如表 15-2 所示。

表　15-2

参　　数	调　整　值
SGA_TARGET	50 GB
DB_CACHE_SIZE	16 GB
DB_KEEP_CACHE_SIZE	8 GB
DB_RECYCLE_CACHE_SIZE	8 GB
SHARED_POOL_SIZE	8 GB
SESSION_CACHED_CURSORS	100

这套调整方案提出后，客户感到十分不解，对于将共享池调整得如此之大，他们觉得存在很大的风险，也和老白发生了多次争执，甚至暗示如果系统出现问题，我们必须承担全责。当时优

化小组的成员对此也捏了一把汗，不过老白却十分淡定，并表示如果将共享池设置为 10 GB 甚至更大，效果会更好。

优化实施后，系统性能得到了明显的提升，但在每天业务最高峰时还是会出现一些共享池的闩锁争用：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                          % Total
Event                        Waits    Time(s)    Avg Wait(ms)  Call Time   Wait Class
----------------------- ------- ----------- -------------- ---------- ------------
CPU time                              8,695                       39.8
latch: library cache     23,000      5,819            253         26.6     Concurrency
latch: shared pool       18,650      4,790            257         21.9     Concurrency
latch free                6,538      1,842            282          8.4           Other
latch: row cache objects  2,601        709            272          3.2     Concurrency
-----------------------------------------------------------------------------------
```

从上面的数据来看，问题似乎并没有解决，反而更严重了。当时客户对优化效果提出了质疑，而且希望我们将系统恢复到原来的配置。但老白却认为优化的方向并没有错，反而应该进一步加大共享池，将其设置为 12 GB，针对此观点，大家又出现了意见的不一致，因此并没有马上实施。但在两天后，共享池争用逐渐消失了。

```
Cache Sizes
~~~~~~~~~~~~~~~~~                 Begin      End
              Buffer Cache:     17,568M    17,568M      Std Block Size:         16K
           Shared Pool Size:     13,056M    13,056M          Log Buffer:     47,016K

Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                 Redo size:             705,367.85             21,262.35
             Logical reads:              63,030.79              1,899.98
             Block changes:               3,570.40                107.62
            Physical reads:                 739.17                 22.28
           Physical writes:                  52.17                  1.57
                User calls:               4,294.22                129.44
                    Parses:               1,460.07                 44.01
               Hard parses:                  88.11                  2.66
                     Sorts:                 370.04                 11.15
                    Logons:                   0.20                  0.01
                  Executes:               1,583.99                 47.75
              Transactions:                  33.17

 % Blocks changed per Read:     5.66    Recursive Call %:     71.00
Rollback per transaction %:     3.30       Rows per Sort:    107.82

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:     99.98       Redo NoWait %:    100.00
               Buffer Hit %:    100.11    In-memory Sort %:    100.00
              Library Hit %:     95.79        Soft Parse %:     93.97
         Execute to Parse %:      7.82         Latch Hit %:     99.74
Parse CPU to Parse Elapsd %:     67.31     % Non-Parse CPU:     90.58
```

```
 Shared Pool Statistics        Begin    End
                               ------  ------
             Memory Usage %:   86.88    85.67
    % SQL with executions>1:   45.53    45.35
  % Memory for SQL w/exec>1:   49.76    46.87

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                % Total       Wait
Event                           Waits  Time(s)  Avg Wait(ms) Call Time      Class
------------------------------ ------- -------- ------------- --------- -----------
CPU time                                 11,319                     2.3
enq: TX - row lock contention   11,886    4,986       419           1.0   Application
row cache lock                 908,103      462         1            .1   Concurrency
log file parallel write        415,266      295         1            .1    System I/O
SQL*Net more data to client    10,503,868  231          0            .0       Network
-----------------------------------------------------------------------------------
```

从上面的 AWR 报告来看，共享池扩大到了 12 GB 多，系统的并发访问量也大幅上升，而共享池的各项争用却大幅减少了，共享池问题得到了本质性的改善。本次优化的初步目标已经达成。

在这样一种充满了风险的情况下，老白是如何作出如此判断，并最终获得成功的呢？在实施初期优化效果不理想的情况下，如果老白不能顶住压力，就只能认为这个系统确实不适合配置过大的共享池，从而选择回退操作。那样的话这次优化就彻底失败了。

正是因为对共享池概念的理解，老白才选择了坚持。这个系统 CPU 资源充足，平时 CPU 使用率不足 20%，内存容量也十分充足，达到了 128 GB。而扩大共享池可能带来的副作用只是管理更大内存需要消耗更多的 CPU 资源，而目前系统的 CPU 达到了 32 核，共享池被分为 7 个子池，哪怕此时配置 14 GB 的共享池，每个子池也不过只有 2 GB 的容量。在 Oracle 8*i* 的系统中，配置 2 GB 的共享池都不会导致特别严重的闩锁争用问题，更何况是大幅增加了 LRU Bucket 数量的 9*i* 及后续版本。老白坚信加大共享池的正面作用远大于负面影响，因此才能够从容应对。

而客户对共享池的认识受到了网络上一些不够严谨的文章的误导，存在一定的误区。另外，他们进行的实验也不够完整，仅测试了 4 GB 共享池下的性能情况，就认定共享池不能超过 4 GB，如果当时他们能够测试到 10 GB，甚至 14 GB 共享池下的性能情况，那么可能得到的就是另外一个结论了。

从上述案例中，我们可以得到几点启示。首先，我们掌握的知识必须是十分准确的，而错误的、不严谨的知识只能带来副作用；其次，这些知识并不需要特别深入，在此案例中，我们不需要真正了解共享池以及闩锁的具体算法，只要掌握共享池的基本原理就足以作出合理的判断了；最后，有时实验并不能完全说明问题，因为实验的场景、范围可能存在缺陷，根据这样的实验得出的结论其准确性会大打折扣。

在本节中，老白并没有讲述如何将 Oracle 的理论和实践完美结合，因为这是不可能完成的工作，Oracle 的知识点不计其数，实践技巧更是五花八门。老白只能通过具体案例让大家理解如何运用自己的知识，实现相对的结合。

Part 3 第三部分

典型案例篇

典型案例是学习维护技巧十分重要的手段，为了让大家能够对本书介绍的理论和方法有更深的理解，老白精心挑选了 10 多个案例，这些案例都有各自不同的精彩之处，按照本书的内容，可分为 RAC 故障分析、性能优化、故障处理等几个章节。

大家在学习这些案例时，如果对照前面章节所介绍的基础知识和方法，就能对本书的思想有更深入的理解。学习案例需要举一反三，这样才能对今后的工作有所帮助。如果只是照本宣科，那么这些案例也就没有什么价值了。

第 16 章

RAC 故障分析

16

RAC 故障主要集中在集群故障、RAC 实例故障、RAC 性能问题等几个方面，很多 DBA 缺乏 RAC 方面的分析经验，这样就很难找到问题的根本原因。

本章的目的是让读者了解 RAC 分析的方法和思路，通过学习这些简单的案例，我们就可以掌握 RAC 故障分析的一般方法。

16.1 LOG_ARCHIVE_MAX_PROCESS 导致的 RAC 脑裂

今天一上班就接到客户的电话，RAC 中的一个数据库实例登不上去了。于是，我马上通过 VPN 连接系统，首先检查了所有节点的 ALERT LOG 日志，在 RAC1 节点的 ALERT LOG 日志中，可以看到：

```
Tue Dec 14 07:10:39 2010
ARC7: Attempting destination LOG_ARCHIVE_DEST_2 network reconnect (3113)
ARC7: Destination LOG_ARCHIVE_DEST_2 network reconnect abandoned
PING[ARC7]: Error 3113 when pinging standby exp5_physical_std.
Tue Dec 14 07:40:45 2010
IPC Send timeout detected.Sender: ospid 1672116
Receiver: inst 2 binc 344730067 ospid 198544
Tue Dec 14 07:41:11 2010
IPC Send timeout to 1.3 inc 80 for msg type 65516 from opid 9
Tue Dec 14 07:43:52 2010
Evicting Instance 2 from cluster
Tue Dec 14 07:44:35 2010
Waiting for Instances to leave:2
Tue Dec 14 07:44:55 2010
Waiting for Instances to leave:2
Tue Dec 14 07:45:15 2010
Waiting for Instances to leave:2
Tue Dec 14 07:45:35 2010
Waiting for Instances to leave:2
Tue Dec 14 07:45:47 2010
IPC Send timeout detected.Sender: ospid 1253834
Receiver: inst 2 binc 344730053 ospid 148480
Tue Dec 14 07:45:55 2010
Waiting for Instances to leave:2
Tue Dec 14 07:46:10 2010
```

```
IPC Send timeout to 1.1 inc 80 for msg type 73 from opid 7
Tue Dec 14 07:46:15 2010
Waiting for Instances to leave:2
Tue Dec 14 07:46:35 2010
Waiting for Instances to leave:2
Tue Dec 14 07:46:55 2010
Waiting for Instances to leave:2
Tue Dec 14 07:47:15 2010
Tue Dec 14 07:53:31 2010
Trace dumping is performing id=[cdmp_20101214075331]
Tue Dec 14 07:53:34 2010
Reconfiguration started (old inc 80, new inc 84)
List of nodes:
 0 2
 Global Resource Directory frozen
 * dead Instance detected - domain 0 invalid = TRUE
 Communication channels reestablished
 * domain 0 not valid according to Instance 2
Tue Dec 14 07:53:34 2010
 Master broadcasted resource hash value bitmaps
 Non-local Process blocks cleaned out
Tue Dec 14 07:53:34 2010
 LMS 1: 59 GCS shadows cancelled, 6 closed
Tue Dec 14 07:53:34 2010
 LMS 4: 105 GCS shadows cancelled, 9 closed
Tue Dec 14 07:53:34 2010
 LMS 2: 100 GCS shadows cancelled, 7 closed
Tue Dec 14 07:53:34 2010
 LMS 0: 114 GCS shadows cancelled, 11 closed
Tue Dec 14 07:53:34 2010
 LMS 5: 90 GCS shadows cancelled, 13 closed
Tue Dec 14 07:53:34 2010
 LMS 3: 113 GCS shadows cancelled, 6 closed
 Set master node info
 Submitted all remote-enqueue requests
 Dwn-cvts replayed, VALBLKs dubious
 All grantable enqueues granted
 Post SMON to start 1st pass IR
```

不难看出，在时间点 7:10:39 出现了 DATAGUARD 归档故障的报错，这个错误很可能和实例宕掉有关。于是我让客户确认 DATAGUARD 是否存在问题。经过检查发现，这个库每天早上都会停止 DATAGUARD，并将 DATAGUARD 的数据文件通过 LVM 同步到远端的一个存储中，供其他业务系统使用。出现报错的时间点和上述操作的时间是吻合的。

从上述日志来看，RAC1 节点和 RAC2 节点通信时出现了 IPC 超时，系统进行了仲裁，将 RAC2 节点驱逐了，接下来我们检查了 RAC2 节点的 ALERT LOG 日志：

```
Tue Dec 14 07:11:42 2010
ARC5: Attempting destination LOG_ARCHIVE_DEST_2 network reconnect (3113)
ARC5: Destination LOG_ARCHIVE_DEST_2 network reconnect abandoned
PING[ARC5]: Error 3113 when pinging standby exp5_physical_std.
Tue Dec 14 07:17:26 2010
Process q000 died, see its trace file
```

```
Tue Dec 14 07:17:26 2010
ksvcreate: Process(q000) creation failed
Tue Dec 14 07:24:11 2010
ksvcreate: Process(q000) creation failed
Tue Dec 14 07:44:16 2010
ksvcreate: Process(q002) creation failed
Tue Dec 14 07:46:46 2010
MMNL absent for 1201 secs; Foregrounds taking over
Tue Dec 14 07:46:46 2010
MMNL absent for 1201 secs; Foregrounds taking over
Tue Dec 14 07:50:17 2010
IPC Send timeout detected. Receiver ospid 198544
Tue Dec 14 07:50:20 2010
Errors in file /dba/app/oracle/admin/szjf/bdump/szjf2_lms2_198544.trc:
Tue Dec 14 07:52:57 2010
IPC Send timeout detected. Receiver ospid 148480
Receiver is waiting for a latch dumping latch state for receiver -17260
Tue Dec 14 07:52:57 2010
Errors in file /dba/app/oracle/admin/szjf/bdump/szjf2_lms0_148480.trc:
Tue Dec 14 07:53:29 2010
ARC9: terminating Instance due to error 481
Tue Dec 14 07:53:29 2010
Errors in file /dba/app/oracle/admin/szjf/bdump/szjf2_lmon_111882.trc:
ORA-29740: evicted by member 2, group incarnation 82
```

在上述日志中，我发现 RAC2 节点在时间点 7:11:42 也出现了类似的归档故障，随后在时间点 7:50:17 出现了 LMS2 进程的 IPC 超时，在时间点 7:52:57 出现了 LMS0 进程的 IPC 超时，在时间点 7:53:29 由于节点被驱逐，所以 ARC9 进程终止了实例。

在后面的 ALERT LOG 日志中，我发现此时数据库已经在进行重启：

```
Successful mount of redo thread 2, with mount id 1617700910
Tue Dec 14 07:55:29 2010
Database mounted in Shared Mode (CLUSTER_DATABASE=TRUE)
Completed: ALTER DATABASE   MOUNT
Tue Dec 14 07:55:30 2010
ALTER DATABASE OPEN
Picked broadcast on commit scheme to generate SCNs
```

经过沟通，得知数据库已经重启了一段时间，不过一直停留在 SCN 广播通信上，已经有 30 多分钟没有任何响应了。

在 10*g* 版本中，SCN 广播是实时模式的，一般延时很小，因此这个环节不会持续较长时间，但目前系统已经等待了 30 多分钟，仍未完成数据库的打开，这是十分不正常的。由于 9 点是公司的上班时间，为了不影响整个公司的运作，我决定再次重启这个实例，不过重启过程还是没有成功，仍然停留在上回的那个地方。

```
Database mounted in Shared Mode (CLUSTER_DATABASE=TRUE)
Completed: ALTER DATABASE   MOUNT
Tue Dec 14 08:32:09 2010
ALTER DATABASE OPEN
Picked broadcast on commit scheme to generate SCNs
```

到底是什么原因导致实例无法打开呢？我首先检查了 REDO LOG 日志的状态，通过查询 V$LOG，发现了一个异常，2 号实例中存在一个处于活动状态的 REDO LOG 日志，这个日志是未归档状态的。

```
GROUP# THREAD# SEQUENCE#    BYTES MEMBERS ARC STATUS
------ ------- --------- -------- ------- --- -------
     9       2      8690 52428800       2 YES  CURRENT
    10       2      8686 52428800       2 YES INACTIVE
    11       2      8684 52428800       2 YES INACTIVE
    12       2      8685 52428800       2 YES INACTIVE
    13       2      8686 52428800       2 YES INACTIVE
    14       2      8687 52428800       2 YES INACTIVE
    15       2      8688 52428800       2 YES INACTIVE
    16       2      8689 52428800       2  NO   ACTIVE
```

于是，我在 Metalink 上搜索了“Picked broadcast on commit scheme to generate SCNs”，搜索结果如图 16-1 所示。

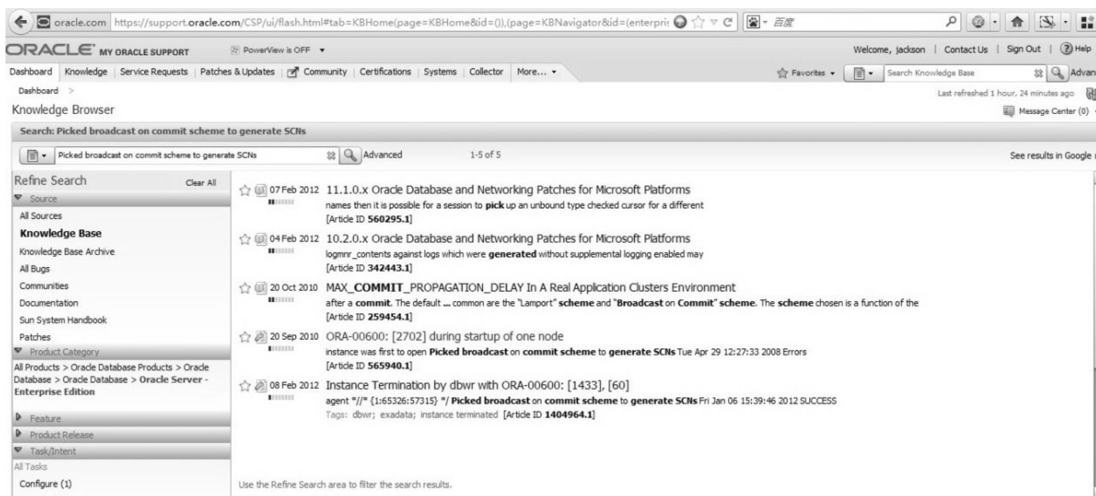


图　16-1

其中，ORA-00600: [2702] during startup of one node [ID 565940.1]虽然和我们目前碰到的情形不完全一致，但是具有一定的相似性。通过分析这篇文档，我决定立即关闭所有的实例，进行一次数据库恢复操作，然后再启动实例。

通过和业务部门以及上级主管部门的协商，我们决定在 8 点 40 分关闭其他两个正常工作的实例，在进行数据库恢复操作后重启数据库。

```
Tue Dec 14 09:02:38 2010
ALTER DATABASE RECOVER  database
Media Recovery Start
 parallel recovery started with 16 processes
Tue Dec 14 09:02:54 2010
```

```
Recovery of Online Redo Log: Thread 2 Group 31 Seq 104077 Reading mem 0
  Mem# 0: /dev/rv_redo2_13
Tue Dec 14 09:02:54 2010
Recovery of Online Redo Log: Thread 1 Group 7 Seq 133118 Reading mem 0
  Mem# 0: /dev/rv_redo1_03
Tue Dec 14 09:02:54 2010
Recovery of Online Redo Log: Thread 3 Group 43 Seq 69756 Reading mem 0
  Mem# 0: /dev/rv_redo3_11
Tue Dec 14 09:03:01 2010
Recovery of Online Redo Log: Thread 3 Group 44 Seq 69757 Reading mem 0
  Mem# 0: /dev/rv_redo3_12
Tue Dec 14 09:03:03 2010
Recovery of Online Redo Log: Thread 1 Group 8 Seq 133119 Reading mem 0
  Mem# 0: /dev/rv_redo1_04
Tue Dec 14 09:03:09 2010
Recovery of Online Redo Log: Thread 1 Group 9 Seq 133120 Reading mem 0
  Mem# 0: /dev/rv_redo1_05
Tue Dec 14 09:03:10 2010
Recovery of Online Redo Log: Thread 2 Group 32 Seq 104078 Reading mem 0
  Mem# 0: /dev/rv_redo2_14
Tue Dec 14 09:03:12 2010
Recovery of Online Redo Log: Thread 3 Group 45 Seq 69758 Reading mem 0
  Mem# 0: /dev/rv_redo3_13
Tue Dec 14 09:03:16 2010
Recovery of Online Redo Log: Thread 1 Group 10 Seq 133121 Reading mem 0
  Mem# 0: /dev/rv_redo1_06
Tue Dec 14 09:03:25 2010
Recovery of Online Redo Log: Thread 1 Group 11 Seq 133122 Reading mem 0
  Mem# 0: /dev/rv_redo1_07
Tue Dec 14 09:03:30 2010
Recovery of Online Redo Log: Thread 2 Group 33 Seq 104079 Reading mem 0
  Mem# 0: /dev/rv_redo2_15
Tue Dec 14 09:03:31 2010
Recovery of Online Redo Log: Thread 3 Group 46 Seq 69759 Reading mem 0
  Mem# 0: /dev/rv_redo3_14
Tue Dec 14 09:03:35 2010
Recovery of Online Redo Log: Thread 1 Group 12 Seq 133123 Reading mem
Tue Dec 14 09:03:41 2010
Recovery of Online Redo Log: Thread 1 Group 13 Seq 133124 Reading mem 0
  Mem# 0: /dev/rv_redo1_09
Tue Dec 14 09:03:45 2010
Recovery of Online Redo Log: Thread 1 Group 14 Seq 133125 Reading mem 0
  Mem# 0: /dev/rv_redo1_10
Tue Dec 14 09:03:46 2010
Recovery of Online Redo Log: Thread 3 Group 47 Seq 69760 Reading mem 0
  Mem# 0: /dev/rv_redo3_15
Tue Dec 14 09:03:50 2010
Recovery of Online Redo Log: Thread 1 Group 15 Seq 133126 Reading mem 0
  Mem# 0: /dev/rv_redo1_11
Tue Dec 14 09:03:52 2010
Recovery of Online Redo Log: Thread 1 Group 16 Seq 133127 Reading mem 0
  Mem# 0: /dev/rv_redo1_12
```

```
Tue Dec 14 09:03:54 2010
Recovery of Online Redo Log: Thread 1 Group 17 Seq 133128 Reading mem 0
  Mem# 0: /dev/rv_redo1_13
Tue Dec 14 09:03:54 2010
Recovery of Online Redo Log: Thread 3 Group 48 Seq 69761 Reading mem 0
  Mem# 0: /dev/rv_redo3_16
Tue Dec 14 09:03:57 2010
Recovery of Online Redo Log: Thread 1 Group 18 Seq 133129 Reading mem 0
  Mem# 0: /dev/rv_redo1_14
Tue Dec 14 09:03:59 2010
Recovery of Online Redo Log: Thread 1 Group 19 Seq 133130 Reading mem 0
  Mem# 0: /dev/rv_redo1_15
Tue Dec 14 09:04:03 2010
Recovery of Online Redo Log: Thread 1 Group 20 Seq 133131 Reading mem 0
  Mem# 0: /dev/rv_redo1_16
Tue Dec 14 09:04:03 2010
Recovery of Online Redo Log: Thread 3 Group 5 Seq 69762 Reading mem 0
  Mem# 0: /dev/rv_redo3_01
Tue Dec 14 09:04:04 2010
Media Recovery Complete
Completed: ALTER DATABASE RECOVER  database
```

RECOVER DATABASE 操作正常结束，然后我们也正常地打开了数据库。完成 RAC1 节点的恢复操作后，重启 RAC2 节点，也是先以 STARTUP MOUNT 方式启动，确认加载正常后再打开数据库。接下来，使用同样的方法处理 RAC3 节点。终于，在 9 点 15 分所有的节点都正常打开了，系统全部恢复正常。由于 RAC1 节点在 9 点 5 分左右正常启动，主系统在 9 点 10 分左右完全恢复，因此本次故障虽然没能在 9 点之前完成恢复，但是已经将影响降到了最低，可以算是一次较为成功的故障恢复。

完成系统恢复后，我们马上开始诊断问题出现的原因。通过 NMON 数据我们发现故障发生前 RAC2 节点的 CPU 使用率突然从 30%左右暴增，如图 16-2 所示。

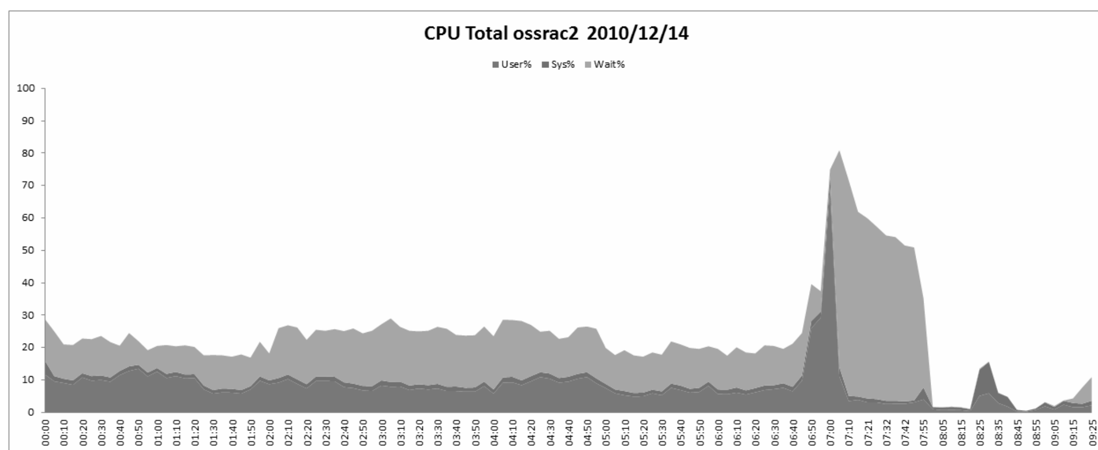


图　16-2

从图中可以看出，CPU 使用率从不到 10%增长到接近 90%，然后出现了较为严重的 IOWAIT，再往后数据库就宕掉了。下面来看一看这个时间段内的磁盘情况，如表 16-1 所示。

表 16-1

时 间	HDISK1	HDISK0	CD0	HDISK10
7:00:27	7.4	7.5	0	13.8
7:05:55	101	100	0	0.5
7:10:56	99.3	99.1	0	0.3
7:16:14	99.4	99.7	0	0.2
7:21:24	101	101	0	0.3
7:27:07	99.4	99.8	0	0.3
7:32:11	97.6	97.6	0	0.3
7:37:23	99.5	99.6	0	0.3
7:42:32	99.7	99.8	0	0.2
7:48:55	101	101	0	0.1
7:55:15	75.1	77.1	0	0.2
8:00:15	2.9	5.2	0	0.4

可以看出，HDISK1 和 HDISK0 从时间点 7:05:55 开始突然变得繁忙起来，直到时间点 7:55:15 才降下来。如果系统盘繁忙，我们的第一怀疑对象就是换页。于是我们查看了换页（page）情况，如图 16-3 所示。

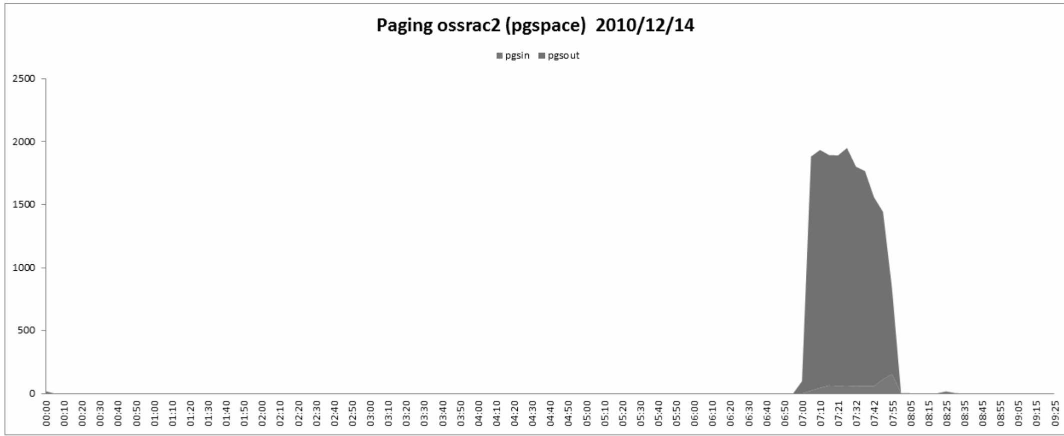


图 16-3

从这张图来看，在系统盘繁忙的这段时间内，确实出现了严重的换页。这个现象可以解释为什么 RAC1 节点会对 RAC2 节点发起驱逐操作，由于换页频繁，CPU 被耗尽，因此导致 RAC2 节点被驱逐。下一步需要了解为什么会出现 CPU 突增的现象。对照 NMON 报告中 CPU 突增的时间点，在查看 ALERT LOG 日志后我们发现，由于 DATAGUARD 停止接收日志，所以远程归

档失败，此时大量归档进程被启动，不断地进行重试（retry）操作，而当时正好物理内存空闲空间较小，又出现了大量的换页，因此就导致了故障。为防止归档进程再次消耗过多资源，我建议客户将归档进程的最大数设置为 3：

```
ALTER SYSTEM SET log_archive_max_processes=3 SCOPE=BOTH SID='*';
```

至此，这个问题似乎已经解决了，我也根据处理的过程向客户提交了故障处理报告，不过总是觉得好像还有些问题没有搞清楚。于是，我在周末又查看了这个案例的文档，突然发现了一个新的情况，如图 16-4 所示。

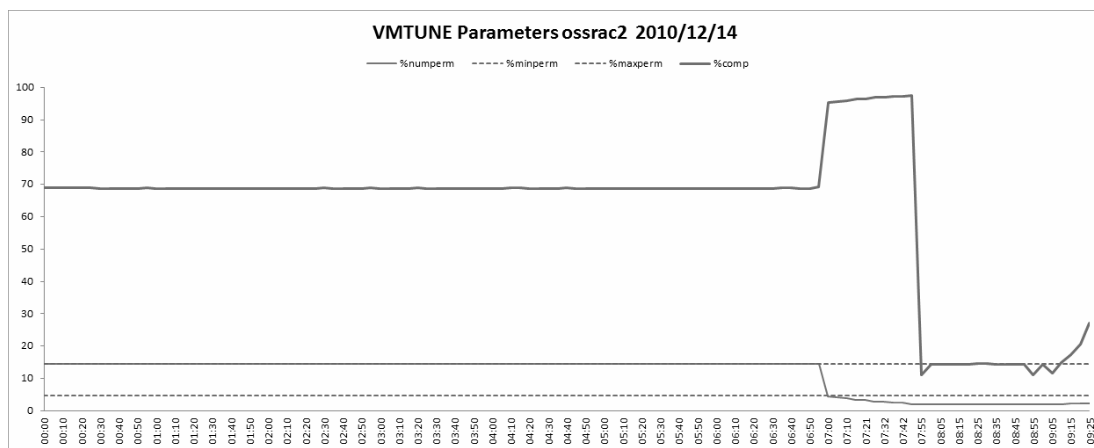


图　16-4

从这张图中可以看出，在故障发生前，物理内存的使用率是 70%，由于 MAXPERM%参数设置为 20%，因此这部分物理内存就被 NOCOMP 内存使用了。在大量归档进程启动时，物理内存突然被耗尽了，此时 NOCOMP 内存开始释放，这个过程持续了 20 多分钟。看来还需要对这个系统做进一步的调整，为了确保今后系统的稳定运行，将 MAXPERM%参数的值减少到 10%是很有必要的。

16.2　RAC 系统故障的处理过程

晚上 7 点左右，突然接到客户的求助电话，数据库出现了一些问题，有一个节点重启了，而重启后系统一直比较慢，需要我帮忙分析问题原因并尽快恢复系统。于是，我马上登录系统进行分析，按照惯例，首先检查了 ALERT LOG 日志。日志中并无异常，只是日志切换的频率似乎比平时低了，从以往的经验来看，这种现象说明系统的性能存在问题，导致单位时间的 REDO 量出现了较大的下降。

在这种情况下，马上进行 HANGANALYZE 分析是十分必要的。通过 HANGANALYZE 报告可以看出，节点 1 中大量的会话被挂起了，并且存在大量 GC BUFFER BUSY WAIT 等待的会话，

杀掉其中几个阻塞严重的会话后，节点 1 恢复正常。

处理完节点 1 后，节点 2 的情况并未得到改善，运行仍然缓慢。于是，我使用同样的方法对节点 2 进行了 HANGANALYZE 分析，发现节点 2 也存在大量挂起的会话。这时已经是晚上的 7 点 30 分了，由于从晚上 8 点开始，节点 2 上有一个十分重要的批处理要运行，在和客户沟通后，我决定先恢复应用再分析问题，并建议马上重启这个实例。实例 2 重启后，系统恢复正常。

此时就需要进一步分析问题的原因了。我再次仔细检查了 ALERT LOG 日志，发现这个 RAC 中的节点 3 在时间点 18:52 附近出现了故障，并且进行了自动重启，之后就出现了系统挂起的现象。造成此现象主要原因是节点 1 在帮助节点 3 进行实例恢复时出现了大量的全局热块冲突，从而导致了问题。

```
Fri Aug 20 18:52:31 2010
IPC Send timeout detected.Sender: ospid 217768
Receiver: inst 3 binc -632201635 ospid 422018
Fri Aug 20 18:52:40 2010
IPC Send timeout to 2.8 inc 6 for msg type 73 from opid 8
Fri Aug 20 18:52:40 2010
Communications reconfiguration: Instance_number 3
Fri Aug 20 18:52:50 2010
Trace dumping is performing id=[cdmp_20100820185240]
Fri Aug 20 18:52:57 2010
IPC Send timeout detected.Sender: ospid 177036
Receiver: inst 3 binc -632201634 ospid 357254
Fri Aug 20 18:53:24 2010
IPC Send timeout to 2.7 inc 6 for msg type 65516 from opid 7
Fri Aug 20 18:55:30 2010
IPC Send timeout detected.Sender: ospid 405882
Receiver: inst 3 binc -632201643 ospid 164598
Fri Aug 20 18:55:30 2010
IPC Send timeout detected.Sender: ospid 836012
Receiver: inst 3 binc -632201634 ospid 357254
Fri Aug 20 18:55:30 2010
IPC Send timeout detected.Sender: ospid 926704
Receiver: inst 3 binc -632201642 ospid 438612
Fri Aug 20 18:55:30 2010
IPC Send timeout detected.Sender: ospid 856508
```

从节点 1 的日志中可以看出，在时间点 18:52:31，出现了第一个 IPC 超时报警：

```
IPC Send timeout detected.Sender: ospid 217768
Receiver: inst 3 binc -632201635 ospid 422018
```

这是因为节点 1 与节点 3 的 422018 进程通信出现了故障。另外，节点 2 与节点 3 的 422018 进程也出现了通信故障。于是，下一步的工作重点就是排查 422018 进程的 trace 文件，该文件的详细信息如下：

```
*** 2010-08-20 18:47:22.933
SKGXP_TRACE_FLUSH: output truncated 18 traces lost
SKGXP_TRACE_FLUSH: start
```

```
(422018 <- 475916)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 475916)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 475916)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 495936)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 930222)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 635400)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 357116)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 578732)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 636266)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 491916)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 688192)SKGXPDOCON:can't accept new connections under timed wait context
(422018 <- 525226)SKGXPDOCON:can't accept new connections under timed wait context
SKGXP_TRACE_FLUSH: end
*** 2010-08-20 18:53:23.983
KJM_HISTORY: RCVR STALL OP(14) context 32 elapsed 342061334 us
KJM HIST LMS7:
  14:32:342061334 1:1 14:32:84561 1:0 14:32:52 1:0 14:32:49819 1:11 14:32:78688 1:0
  14:36:52 1:0 14:32:78 1:0 14:32:63 1:1 14:32:48 1:0 14:32:79 1:0
  14:32:71 1:0 14:32:73 1:0 14:32:19 1:0 14:36:29 1:0 14:32:22 1:0
  14:32:40 1:0 14:32:136066 1:0 14:32:43 1:0 14:32:126240 1:0 14:32:78 1:0
  12:20:62 7:8 6:28 10:0 17:4 16:15 15:1 14:32:88296 1:0 14:32:87
  1:0 14:32:73 1:0 14:32:24 1:1 14:36:35 1:0 14:32:33 1:1 14:32:41
  1:0 14:32:62 1:0 14:32:13172
----------------------------------------
SO: 7000007023e8670, type: 4, owner: 7000007042bbf60, flag: INIT/-/-/0x00
(session) sid: 1313 trans: 0, creator: 7000007042bbf60, flag: (51) USR/- BSY/-/-/-/-/-
            DID: 0003-000E-00000003, short-term DID: 0000-0000-00000000
            txn branch: 0
            oct: 0, prv: 0, sql: 0, psql: 0, user: 0/SYS
  last wait for 'latch free' blocking sess=0x0 seq=57724 wait_time=8333 seconds since
wait started=352
          address=70000001000f4b0, number=56, tries=439
  Dumping Session Wait History
   for 'latch free' count=1 wait_time=8333
          address=70000001000f4b0, number=56, tries=439
   for 'latch free' count=1 wait_time=292977
          address=70000001000f4b0, number=56, tries=438
   for 'latch free' count=1 wait_time=292977
          address=70000001000f4b0, number=56, tries=437
   for 'latch free' count=1 wait_time=292977
```

从 trace 文件中可以看出，节点 3 的 LMS 进程出现了闩锁等待，该闩锁的编号是 56，通过查询 V$LATCHNAME 视图，我发现这个闩锁是 OS process: request allocation。而之前的消息说明接收新连接超时。


```
(422018 <- 475916)SKGXPDOCON:can't accept new connections under timed wait context
```

SKGXDOCON 是 RAC INTERCONNECT 通信层面的模块。这说明在故障期间，RAC INTERCONNECT 通信出现了异常。于是，我检查了 NMON 报告中的 NET 通信，如图 16-5 所示。

上面的数据没有出现异常，因此不是网络阻塞导致了问题。接下来，进一步分析 CPU 等资源的使用情况，如图 16-6 所示。

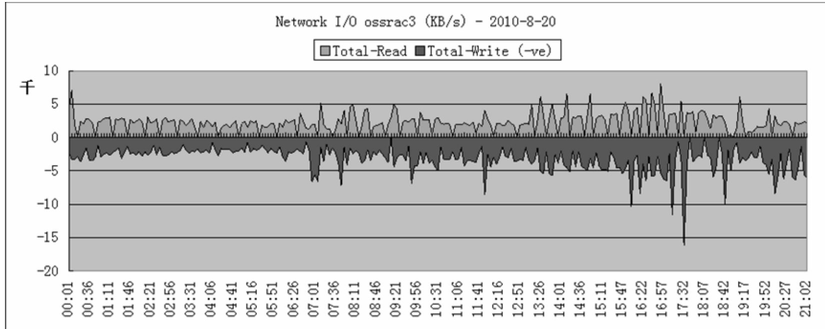


图　16-5

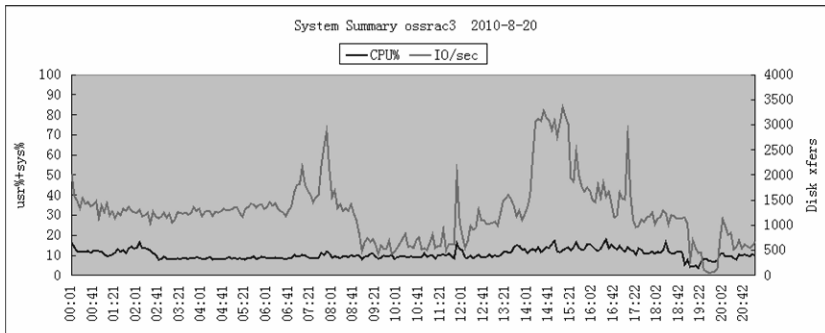


图　16-6

在这张图中我也没有发现异常。在时间点 18:00 后，系统负载并没有增加，反而 I/O 有较大的下降，这是因为下班后使用系统的人在减少。至此，分析工作似乎陷入了僵局，于是我询问了 DBA，在故障前是否进行过特殊操作。后来得知在 6 点 40 分左右，DBA 曾通过 SCP 在两个节点间复制了一个几十 GB 的文件，但开始没多久，系统就宕机了。这条线索让我眼前一亮，立即检查了 NMON 报告中的换页信息，如图 16-7 所示。

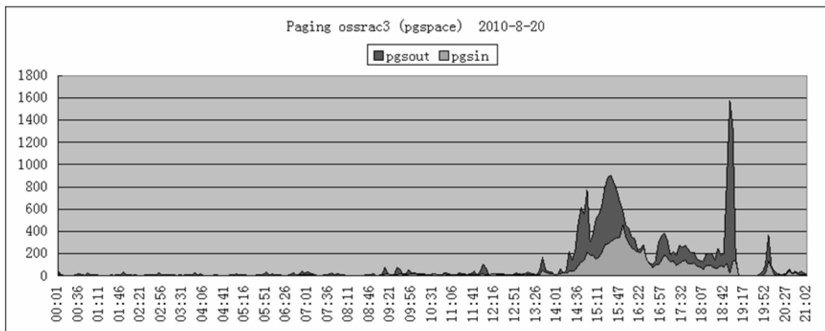


图　16-7

我们可以看到一个明显的问题，在时间点 18:42 附近，系统出现了十分严重的换页现象。这足以导致 LMS7 进程无法获取到 CPU 资源，从而产生网络通信超时。经过检查，我发现这台服务器的 MAXPERM%参数设置为 80%。就这样绕了一大圈，问题又回到了 MAXPERM%参数，看来著名的 IBM 红皮书真是害人不浅啊。

16.3　三天两次严重故障

这是一套 4 节点的 RAC 系统，4 月 19 日，由于一个新上线的业务模块出现问题，导致一个节点负载特别高。业务部门发现问题后，马上杀掉了这个应用。但在故障处理期间，这个实例宕机了。由于这套系统并未根据 SERVICE_NAME 划分应用的实例，而是直接通过 IP 地址连接的。因此该实例宕机后，其应用无规律地转移到了其他节点，而这个节点上存在一些很大的批处理作业，因此应用故障转移（failover）后由于 DRM 机制的存在，出现了严重的争用，从而导致系统的性能下降，其中有一个实例基本无法使用。客户的 DBA 发现问题后，想关闭并重启这个实例，但重启了 10 多分钟也没能打开。于是才联系我，希望我马上到现场帮他们处理。

在我到达现场前，那个实例已经打开了。但由于故障转移的问题，其他实例又开始出现问题了。于是我建议先暂停应用，再重启实例，将有问题的实例全部重启一遍。经过 1 个多小时的工作，我们终于在下午 4 点多恢复了系统的运行。没有影响 5 点到 7 点的业务高峰。

系统恢复后，我们简单地分析了故障的原因。毫无疑问，某个实例宕机后应用的无序切换是导致所有节点故障的主要原因。这是由于应用没有使用 SERVICE_NAME 所导致的，目前暂时还没有解决方案，如果修改连接池的配置，将涉及 300 多个配置项，调整工作量太大，开发部门也拒绝修改。本次故障的处理过程超过两个小时，是因为系统启用了 DRM，导致实例重启十分缓慢。为了确保这种情况不会再次出现，我建议他们关闭 DRM，但由于完成此操作需要重启所有节点，所以我们仅修改了 SPFILE 的参数，等待下一个停机窗口进行重启。

本来以为这次故障算是处理完了，由于在整个过程中业务没有完全中断，关键业务也能时断时续地进行，因此在客户那里并没有造成太大的影响。所以客户对故障分析报告催得并不是很紧。但正在我考虑何时去递交分析报告的时候，突然出大事了。

两天后，也就是 4 月 21 日下午 1 点 10 分左右，系统响应时间变慢，4 个实例均出现挂起现象。我到达现场时，系统已经从较慢变为基本不可用了，客户的 DBA 正在尝试关闭所有的实例。

不过在关闭数据库的时候遇到了麻烦，在服务器上使用 sysdba 账号无法登录数据库，因此也就无法正常关闭实例，只能通过 kill 命令杀掉 SMON 进程强制关闭数据库。

但这个过程并不顺利，除了通过 KILL SMON 命令成功关闭节点 4 外，其他节点的 SMON 进程均无法杀掉，最终我们只能选择重启服务器。在未查明故障原因前，为了使数据库重启后不再出现类似问题，我建议对已经关闭实例的节点 4 也进行重启。

重启后，系统恢复正常，由于事先修改了 DRM 的配置，因此本次重启的时间大幅缩短。关闭 DRM 的设想是基于本系统做了一定程度的应用分区，两个实例中共享数据一般只有一个节点

变更，其他节点都是只读状态，平时业务高峰期4个节点之间的RAC INTERCONNECT通信流量也控制在5～8 MB之间。关闭DRM可以使系统更为稳定，不会因为应用切换错误而引发严重的性能问题。

由于3天内发生了两次重大的系统故障，因此客户的领导要求彻查此事。19日的故障被推到了应用上，勉强也说得过去，不过此故障虽然是应用导致的，但DRM也是推波助澜的主要因素。如果DRM没有启用，就算系统会变慢，也不至于出现不可用的情况。

下午的会议中，我简单地向数据中心的领导汇报了故障的处理情况。数据中心主任希望我们能够确定问题已经解决，并且确保类似的宕机故障不会再次出现。我考虑了一下告诉他，目前问题还没有定位，所以我无法保证问题已经解决了。由于4个节点都出现了故障，因此故障定位相对简单，但由于需要分析的资料很多，整个过程可能需要3～5天的时间。我们已经关闭了DRM，但是否奏效还有待观察，在此期间，我们必须加强对系统的监控。

会议结束后，我们立即开始对本次故障进行分析。首先我和客户的运维小组一起明确了本次问题分析工作的主要内容，包括小型机、存储、操作系统、网络和数据库几个层面，其中，存储和小型机的日志需要由系统组去分析，网络日志由网络组进行分析，而我们的重点工作是分析操作系统和数据库。

很快系统组就有了反馈信息，小型机和存储并无报错，网络组的排查也没有发现任何疑点。我检查了操作系统和HACMP程序的日志，也没有发现明显的错误信息。另外，也没有在故障点前后的crsd和cssd日志中发现有价值的线索。于是我们的分析重点就放在数据库的日志分析上了。

ALERT LOG日志中并没有有价值的信息，PMON、SMON、LMD、LMS进程的日志中也没有疑点。幸运的是，4个节点的diag中都存在一个系统状态转储，这是在杀掉smon进程时，diag进行的转储操作。于是我们通过ass109.awk脚本对这4个trace文件进行了分析。得到的报告让我们大吃一惊，4个节点的块状态信息都是很长的一串，而不是通常的几个或十几个。

```
                    Resource Holder State
Enqueue PS-00000001-00000E0F    ??? Blocker
Enqueue US-000000B1-00000000    ??? Blocker
Enqueue CF-00000000-00000000    18: waiting for 'RDBMS ipc message'
Enqueue TX-02040011-000248F0   112: waiting for 'gc buffer busy'
Enqueue TX-00F90016-000ED217    75: waiting for 'gc buffer busy'
Enqueue TX-01EF0023-0002504A   248: waiting for 'gc current request'
Enqueue TX-01D3001B-0002998E   504: 504: is waiting for 75:
Enqueue TX-01EB0000-00024C39   500: 500: is waiting for 75:
Enqueue TX-01D40015-00029004   511: 511: is waiting for 75:
Enqueue TX-01050019-0012807C   408: 408: is waiting for 75:
Enqueue TX-00040000-0026D455    87:  87: is waiting for 75:
Enqueue TX-014A0024-00095BBD   470: 470: is waiting for 75:
Enqueue TX-0171000A-0007263F   510: 510: is waiting for 75:
Enqueue TX-01820020-0005F1B1   464: 464: is waiting for 75:
Enqueue TX-00150025-002F2748   164: waiting for 'gc buffer busy'
Enqueue TX-0006002A-0020C4D7   140: waiting for 'gc buffer busy'
Enqueue TX-002A0015-0027DD88   195: 195: is waiting for 75:
Enqueue TX-00D30010-00200A5D   125: waiting for 'gc buffer busy'
```

```
Enqueue TX-01420025-000AC2F0   212: 212: is waiting for 75:
Enqueue TX-001C0026-00309D9B   383: 383: is waiting for 75:
Enqueue TX-013A0003-000A5786   463: 463: is waiting for 75:
Enqueue TX-01C80011-0002B04C   499: 499: is waiting for 75:
Enqueue TX-00320005-002C1B48   109: waiting for 'gc buffer busy'
Enqueue TX-01AE0001-000372B4   469: 469: is waiting for 75:
Enqueue TX-01A20017-0003D26E   314: 314: is waiting for 75:
Enqueue TX-02020014-00022D0D   227: waiting for 'gc buffer busy'
Enqueue TX-0026002A-001CCA2B   119: waiting for 'gc buffer busy'
Enqueue TX-00F40015-0013229C   377: 377: is waiting for 75:
Enqueue TX-020B0026-0001E8FF   384: 384: is waiting for 75:
Enqueue TX-0183002C-000590C9   107: waiting for 'gc buffer busy'
Enqueue TX-003C000A-002059FF    85:  85: is waiting for 75:
Enqueue TX-01960019-000347EC   466: 466: is waiting for 75:
Enqueue TX-00CC0023-0024A556   223: 223: is waiting for 75:
Enqueue TX-00E2001D-001936FC   153: 153: is waiting for 75:
Enqueue TX-01AB0001-00037A70   100: waiting for 'gc buffer busy'
Enqueue TX-00CA001E-00260EFB   335: 335: is waiting for 75:
Enqueue TX-011A0021-000DDC98   117: waiting for 'gc buffer busy'
Enqueue TX-00EF002F-00168502   380: 380: is waiting for 75:
Enqueue TX-01B70008-0002CC51   376: 376: is waiting for 75:
Enqueue TX-01CA001C-0002D335   336: 336: is waiting for 75:
Enqueue TX-00DF000E-00197E77   225: waiting for 'gc buffer busy'
Enqueue TX-02050028-00022001   490: 490: is waiting for 75:
Enqueue TX-0106001D-00139EA4   256: waiting for 'gc current request'
Enqueue TX-020F000E-0003831F   416: 416: is waiting for 75:
Enqueue PS-00000001-00000E0E    ??? Blocker
Enqueue TX-01D2002A-0002723B   390: 390: is waiting for 75:
Enqueue TX-001B000B-002D7BFE   506: 506: is waiting for 75:
Enqueue TX-00C00005-001B0890   423: 423: is waiting for 75:
Enqueue TX-0176002A-00065BA7   131: waiting for 'gc buffer busy'
Enqueue TX-00F9002F-000ED242    75: waiting for 'gc buffer busy'
Enqueue TX-01C00001-000318E4    58: 58: is waiting for 75:
Enqueue TX-01A5002A-00039637   501: 501: is waiting for 75:
Enqueue TX-01670021-000836C4   186: 186: is waiting for 75:
Enqueue TX-01910019-00046348   496: 496: is waiting for 75:
Enqueue TX-01300020-000AE76D   460: 460: is waiting for 75:
Enqueue TX-01D60018-000239E1   402: 402: is waiting for 75:
Enqueue TX-02110008-0001E839   104: waiting for 'gc buffer busy'
Enqueue TX-01A9000B-0003578E   468: 468: is waiting for 75:
Enqueue TX-0119002C-000D9025   498: 498: is waiting for 75:
Enqueue TX-01F70021-000215D0   426: 426: is waiting for 75:
Enqueue TX-01CD001A-00023733   352: 352: is waiting for 75:
Enqueue TX-00E70020-0020DCB1   216: 216: is waiting for 75:
Enqueue TX-01A6002F-0003C312   165: waiting for 'gc buffer busy'
Enqueue TX-00C20029-001BEBC0   163: 163: is waiting for 75:
Enqueue TX-014E0020-000796DD   333: 333: is waiting for 75:
Enqueue TX-00CF0022-001DFE37   110: 110: is waiting for 75:
Enqueue TX-02030023-00023F0A   150: waiting for 'gc buffer busy'
Enqueue TX-01C50014-0002ED41   378: 378: is waiting for 75:
Enqueue TX-01890018-0004EF95   515: 515: is waiting for 75:
Enqueue TX-003B0021-002090C8   305: 305: is waiting for 75:
```

16

```
Enqueue TX-0160001A-000A5D5F   108: waiting for 'gc buffer busy'
Enqueue TX-01E7000B-00024852   166: 166: is waiting for 75:
Enqueue TX-00D80023-0029E396   482: 482: is waiting for 75:
Enqueue TX-000A000A-007125AB   207: 207: is waiting for 75:
Enqueue TX-019F0023-0004AD96   328: waiting for 'gc current request'
Enqueue TX-012E0022-000AF5D3   439: 439: is waiting for 75:
Enqueue TX-01F80000-00021665   193: waiting for 'gc buffer busy'
Enqueue TX-013B0002-0009BD23   126: waiting for 'gc buffer busy'
Enqueue TX-00BF0018-0019DFA9   201: 201: is waiting for 75:
Enqueue TX-01680000-0008A8B1   226: waiting for 'gc buffer busy'
Enqueue TX-019A0001-00042D84   237: waiting for 'gc buffer busy'
Enqueue TX-01AA001C-00035569   101: waiting for 'gc buffer busy'
Enqueue TX-01B2001D-00038628   458: 458: is waiting for 75:
Enqueue TX-01EC002E-000240DB   475: 475: is waiting for 75:
Enqueue TX-003A0001-001FBA28   222: 222: is waiting for 75:
Enqueue TX-00240029-001F0BB4   385: 385: is waiting for 75:
Enqueue TX-01610016-00099F4F   393: 393: is waiting for 75:
Enqueue TX-01990016-0003FFC3   155: 155: is waiting for 75:
Enqueue TX-00D9000D-0029BEB0   401: 401: is waiting for 75:
Enqueue TX-01A80012-000360FF   210: 210: is waiting for 75:
Enqueue TX-01A10016-0003D84B   130: waiting for 'gc buffer busy'
Enqueue TX-016E0011-00074FAC   561: waiting for 'gc current request'
Enqueue FB-00000006-E70AAF0C   675: waiting for 'log file sync'
Enqueue FB-00000006-89C45E0C   693: waiting for 'log file sync'
Enqueue FB-00000006-E64ABA0D    ??? Blocker
Enqueue FB-00000006-B9CAAB8E   726: waiting for 'log file sync'
Enqueue FB-00000006-B9CAAB8F    ??? Blocker
LOCK: handle=700000b8ee38e28    68:  68: is waiting for Enqueue US-000000B1-00000000
LOCK: handle=700000b8ee38e28    69:  69: is waiting for Enqueue US-000000B1-00000000
LOCK: handle=700000b8ee38e28   513: waiting for 'SQL*Net message from client'
Enqueue FB-00000006-B9CAAB90   753: waiting for 'log file sync'
Enqueue FB-00000006-BA0ABF89    ??? Blocker
```

在这种情况下，逐个排查阻塞数据（blocker）的工作量非常大，而且似乎是吃力不讨好的事情。一般来说，遇到这种情况，应该重点关注几个核心后台进程。通过对 trace 文件的分析，我们发现实例 1 的 CKPT 进程（PROCESS 18）持有 CF 锁，并且在等待 IPC 消息，SMON 进程在等待 DFS LOCK HANDLE 锁。

实例 2 存在大量的 GC BUFFER BUSY 等待事件，在系统状态转储中发现了 1096 个 GC 方面的等待。CKPT 进程在等待 IPC 消息。

实例 3 存在 1600 多个的 GC 等待和 SQ 锁的等待，同时 CKPT 进程持有 CF 锁，并在等待 IPC 消息，SMON 进程在等待 DFS LOCK HANDLE 锁。

实例 4 的持有相对比较简单：

```
                    Resource Holder State
       Latch 70000001000e1c8    ??? Blocker
       Latch 70000001000cea8    ??? Blocker
       Latch 700000b5c300798    ??? Blocker
Enqueue XR-00000004-00000000    18: Self-Deadlock
Enqueue DR-00000000-00000000    ??? Blocker
Rcache object=700000b8a9241f8,  ??? Blocker
```

```
Enqueue CF-00000000-00000000    18: Self-Deadlock
       Latch 700000b5ef19890   526: 526: is waiting for 526: 604: 846: 1077: 1102:
       Latch 700000b5ef19890   604: 604: is waiting for 526: 604: 846: 1077: 1102:
       Latch 700000b5ef19890   846: 846: is waiting for 526: 604: 846: 1077: 1102:
       Latch 700000b5ef19890  1077: 1077: is waiting for 526: 604: 846: 1077: 1102:
       Latch 700000b5ef19890  1102: 1102: is waiting for 526: 604: 846: 1077: 1102:
       Latch 700000b69b3c7f0    ??? Blocker
       Latch 700000b5c3d5c40    ??? Blocker
Enqueue FB-00000073-DD42440A    82: waiting for 'log file sync'
Enqueue TX-00B10029-000AF301    44: 44: is waiting for Latch 70000001000cea8
Enqueue US-000000B1-00000000   325: last wait for 'latch: cache buffers chains'
       Latch 700000b5e196160    ??? Blocker
       Latch 700000b6970fd80    ??? Blocker
       Latch 700000b69e24580    ??? Blocker
       Latch 700000b6c7d4b80    ??? Blocker
```

在上面的数据中我们看到了一条十分抢眼的信息：

```
Enqueue XR-00000004-00000000    18: Self-Deadlock
```

XR 锁居然出现了自死锁，而自死锁的进程是 18 号，也就是 CKPT 进程，通过阅读 CKPT 进程的进程状态转储（process state dump），我们发现是 CF 锁和 XR 锁的自死锁。

从上面的信息来看，似乎实例 4 是锁死所有 4 个节点的"元凶"，其他进程在和实例 4 的 CKPT 进程进行信息同步时产生了锁死的现象。如果真是这样，那么故障出现前后，应该存在大量的 DFS LOCK HANDLE 等待，而且等待的原因应该和 CKPT 进程或者 CF 锁有关。如果我们在处理一套 9*i* 或者更早期版本的数据库，那么就只能依靠猜测或者在 trace 中一点点地查找可能存在的蛛丝马迹。幸运的是，我们目前分析的是一套 10*g* 版本的系统，因此可以通过 ASH 数据进一步定位。

通过检查 ASH 数据，我们发现在故障发生前，DFS LOCK HANDLE 锁等待的现象十分严重，而且所有的 DFS LOCK HANDLE 锁的 ID2 都是 3，根据相关资料（《ORACLE RAC 日记》一书中有关于 DFS LOCK HANDLE 锁的详细介绍，如果大家有兴趣也可以查阅 MOS 上的相关内容，这里就不详细介绍该锁的 ID2 的含义了），我们知道 ID2=3 意味着"DBWR syncronization of SGA with control file"，这是一个和控制文件有关的等待，含义是 DBWR 要同步 SGA 中的控制文件信息以及控制文件上的控制文件信息。

似乎我们已经找到了导致 4 个节点全部出现故障的节点——节点 4，而且也分析清楚了 DFS LOCK HANDLE 的等待原因，即都在等待将 SGA 同步到控制文件。但其实离找到真正的"元凶"还有很长的路。

通过前面的分析我们发现，系统中存在大量的 DFS LOCK HANDLE 等待，这类等待在 RAC 环境中是一种跨实例（cross instance）的等待，主要是在等待其他实例完成某项工作。我们从事故点开始向前分析，在采集大量的 AWR 报告，并进行了更深入地分析后发现，本系统正常情况下半小时采样周期内的 AWR 报告中的 DFS HANDLE LOCK 等待次数大约为 15 000 次，平均等待时间为 1 毫秒；而故障前平均半小时的 DFS LOCK HANDLE 等待次数达到 235 万次以上，平均每次的等待时间为 1 毫秒。数据库重启前的数据如下：

```
                                                                  Avg
                                               %Time  Total Wait    wait     Waits
Event                                  Waits   -outs    Time (s)    (ms)      /txn
---------------------------- ------------- ------ ----------- ------- ---------
DFS lock handle                    2,352,423      .0       1,374       1     422.7
```

数据库重启后的数据如下：

```
                                                                  Avg
                                               %Time  Total Wait    wait     Waits
Event                                  Waits   -outs    Time (s)    (ms)      /txn
---------------------------- ------------- ------ ----------- ------- ---------
DFS lock handle                       15,679      .0          11       1       2.2
```

从这两组数据的对比分析可以看出，19 日数据库重启后，DFS LOCK HANDLE 等待次数均在 200～350 万次之间；而 19 日数据库重启前，DFS LOCK HANDLE 等待次数处于正常水平：

```
                                                                  Avg
                                               %Time  Total Wait    wait     Waits
Event                                  Waits   -outs    Time (s)    (ms)      /txn
---------------------------- ------------- ------ ----------- ------- ---------
DFS lock handle                        9,346      .0           5       1       0.7
```

从 19 日故障前、故障处理完毕，以及 21 日系统重启后的 DFS 相关等待的比较来看，19 日故障处理完毕到 21 日系统故障前的 DFS 相关等待异常，超过正常指标 100 倍以上。从这一点可以看出，在 19 日系统故障后，我们对几个实例都进行了重启，但是没有统一停机重启。在此期间，应用服务器连接池的故障转移出现了紊乱，虽然这种无序的故障转移最终得到了控制，但在跨实例方面仍处于紊乱状态。此后系统一直处于带病工作状态，最终导致平时业务量最小的节点 4 出现了 CF 锁和 XR 锁的自死锁，从而引发了更大的系统故障。

由于节点 4 的 CKPT 进程出现了自死锁，使得节点 4 被挂起，其他节点由于等待节点 4 的 CF 锁也被挂起。导致 CKPT 进程出现自死锁的主要原因是 DFS LOCK HANDLE 等待十分严重。系统中存在大量的 GCS/GES 相关等待。

也就是说，21 日的系统故障和 19 日的故障是有联系的，前者是后者的延续。19 日系统故障后，大多数实例都进行了重启，但是为了确保关键业务能够尽快恢复，各个实例是独立重启的，并未进行统一关闭、统一重启操作。由于实例重启时，应用在大量实例间漂移，因此产生了多次由应用漂移导致的“压死”某个实例的情况。整个实例重启过程持续了数小时，在这段时间内应用漂移现象时有发生。重启完成后，跨实例就一直处于不正常状态，DFS LOCK HANDLE 等待数量从正常的 10 000～15 000 次上升为 2 000 000～3 500 000 次。因此，21 日的故障是问题积累到一定程度时系统的爆发。

另外一个方面，由于跨实例存在问题，从而导致 GCS/GES 性能下降，GCS 等待大量产生，由于应用软件在多实例数据共享方面的开销比较大，而 DRM 功能在 10*g* 版本中又是默认启用的，因此使得 GCS 方面出现了严重的争用，GCS FLUSH 等待大幅增加。这样就引发了 bug 6418420、bug 8215444 这类问题，从而导致 CKPT 出现自死锁，并最终导致 4 个实例全部挂起。

针对上述分析的结论，我们提出了一系列的优化方案：

(1) 建议关闭 DRM（21 日重启前已经实施）。

(2) 优化应用，减少 GCS/GES 争用。

(3) 当某个实例发生故障时，尽可能避免应用在实例间的无序切换，并避免作业在错误的实例上执行。

(4) 目前系统的 I/O 性能不佳，平均 USER I/O 成本在高峰期超过 30 毫秒（正常情况应该小于 10 毫秒），这样会增加 GCS/GES 争用，建议对 I/O 系统进行扩容和优化，从而提高 I/O 子系统的总体性能。

(5) 建议优化 PARALLEL_Instance_GROUPS 参数的设置，避免跨实例的并行查询（故障时节点 3 存在跨四个实例的并行查询），在类似的 OLTP 系统中，跨实例并行查询可能导致严重的性能问题。

(6) 实例 1 上存在较多的 ITL 等待，如果该等待大量存在于 GCS/GES 等待较高的系统中，可能会导致系统挂起。建议对相关表和索引的 INITTRANS 参数进行优化。

(7) 实例 3 上存在较多的 ENQ: SQ 等待，在 RAC 环境中，SQ 等待会增加 GCS/GES 争用的机会，严重时可能会导致系统挂起。建议优化相关的序列参数，加大缓存，尽可能使用 NOORDER 选项。

(8) 建议针对 bug 6418420、bug 8215444 更新补丁。由于 bug 6418420 在 AIX 平台 10.2.0.4 版本上没有 ONE-OFF 补丁，而该修复包含在 PSU 10.2.0.4.1 及以上的版本中，因此，建议将 PSU 升级到 10.2.0.4.4 或者 10.2.0.4.5。

至此，这个案例就介绍完了，在进行了上述调整后，这套系统没有再次出现类似的问题。可能有些朋友看到这里会有些迷惑，如此“轰轰烈烈”的案例怎么就这样平淡地收场了呢，好像也并没有定位到问题的真正根源啊？实际上，上述优化建议主要针对 GC 方面的性能问题，对于如此复杂的宕机故障，想要将其完全定位到一个很小的点上几乎是不可能的。虽然最终的原因可能就是在某个点上，但由于 DBA 的能力、分析投入的成本以及分析所需的资料有限，在绝大多数情况下都无法十分精确地定位是否就是某个 Bug 或者配置导致了问题。不过我们可以大致定位到一个较大范围上，通过对这个范围的分析，提出一套综合性的解决方案，这就是老白经常提到的“组合拳”。

这两个 Bug 可能是导致问题的“元凶”，也可能是“受害者”，简单地将问题推给 Bug 是很不负责任的做法。实际上，出现问题的最终根源还是 GC 和 DRM，因此老白提出了一套综合性的解决方案。从实际的效果来看，对问题点的定位还是比较准确的，根据问题定位所提出的解决方案也是较为合理的，调整后系统的表现就是最好的证明。

第 17 章 ORA-600 故障

ORA-600 是一种十分常见的错误，很多 DBA 都会将它和 Bug 联系在一起。实际上，产生 ORA-600 的原因非常复杂，不能简单地归结到某个 Bug 上，而且并不是所有的 ORA-600 错误都能和 Bug 挂钩。本节将介绍几个关于 ORA-600 的分析案例，通过这些案例，我们可以学习到如何使用 Metalink 来分析这类问题。

17.1 ORA-600 [12700]错误的分析过程

ORA-600[12700]的含义是索引中的 ROWID 指向了一个不存在的行。如果我们通过索引查找某条记录，而索引中的某个 ROWID 指向的行在数据块中并不存在，那么就会出现 ORA-600 [12700]错误。关于 ORA-600 [12700]错误的详细解释可以参考 Metalink 文档 ORA-600 [12700] “Index entry Points to Missing ROWID” [ID 28229.1]，在这里就不过多介绍了。简单地说，ORA-600 [12700]错误是由数据不一致、索引损坏或表数据损坏引起的，比如，实例宕机或者从旧数据文件进行不完全恢复后被强制打开，数据文件中存在不一致。

碰到这类问题该如何解决呢？分析 ORA-600 [12700]错误时，Metalink 文档 Resolving an ORA-600 [12700] error in Oracle 8 and above [ID 155933.1]是一份很好的参考资料。这份文档提出了下面的分析步骤：

(1) 分析包含 ORA-600 [12700]错误的 trace 文件。

(2) 找到损坏的数据库对象，并验证该对象是否损坏。

(3) 找到损坏的索引。

(4) 如果确认是索引损坏，就重建索引。

(5) 如果是由数据块故障导致的，请参考 Note:28814.1 “Handling Oracle Block Corruptions in Oracle7/8/8i” 文档进行处理。

(6) 如果索引和数据均未损坏，那么可能是一致性读存在问题。

首先我们可以在 ALERT LOG 日志中找到 trace 文件，打开 trace 文件可以看到类似的信息：

```
*** 2001-05-29 18:59:53.735
ksedmp: internal or fatal error
ORA-00600: internal error code, arguments: [12700], [2997], [16778259],...
Current SQL statement for this session:
select * from xxxxx where mmmmm='yyy'
```

通过 Current SQL 我们可以定位出现问题的表。不过很多时候，Current SQL 可能是包含多表连接的复杂 SQL，或者 trace 文件根本没有指出 Current SQL，这种情况下，*155933.1* 这份文档后面的部分提供了一个脚本，该脚本能够创建两个用于分析 ORA-600 [12700]的工具——oerr12700 和 oerr12700diag。这两个工具可以帮助我们定位出现故障的行。使用 oerr12700 分析后，我们还可以用 dbv 或者 ANALYZE 命令来分析表数据是否存在问题。

如果表数据没有问题，那么就说明可能是索引损坏了。这时可以使用 ANALYZE TABLE VALIDATE STRUCTURE CASCADE 命令来分析表和索引的结构。要注意的是，如果表很大，分析过程可能需要花费很长的时间，因为分析操作是串行的。如果在生产环境下，而且业务比较繁忙，那么这个操作可能无法运行，或者会对业务产生很大的影响，这种情况下，就可以使用 oerr12700diag 工具了。

```
SQL> set serveroutput on
SQL> execute oerr12700diag(2989,16777219,4294941081)
```

使用 oerr12700 和 oerr12700diag 工具进行存储时,需要输入 3 个参数,这些参数就是 ORA-600 [12700]后面的 3 个参数。

如果确定是索引损坏，一般来说可以直接重建索引；而如果是数据块损坏，就需要进行数据块修复，如果无法修复，可以跳过坏块导出数据，然后重建数据库对象。

如果在索引和表中均未发现问题，那么情况就比较复杂了，可能是出现了某个和一致性读有关的 Bug。如果这个问题是可重现的，那么通过 EVENT 10226 就可以进行更深入的分析。EVENT 10226 的含义是 EVENT: 10226 "trace CR applications of undo for data operations"，可以输出大量 UNDO 和 REDO 的信息，协助分析可能存在的问题。

```
SQL>alter session set events '10226 trace name context forever, level 10' ;
SQL> 执行出问题的模块
```

上面介绍了分析 ORA-600 [12700]问题分析的大体思路，下面通过老白和网友“风”的一段 QQ 对话，来介绍一个真实的问题分析案例。

风 12:17:43

```
老白，我碰到了一个奇怪的问题，你帮我分析一下：
SQL> drop index pk_w_avg;
drop index pk_w_avg
           *
ERROR at line 1:
ORA-02429: cannot drop index used for enforcement of unique/primary key
SQL> alter table w_avg drop primary key;
alter table w_avg drop primary key
*
ERROR at line 1:
ORA-00600: internal error code, arguments: [12700], [25], [4228580], [178], [],
[], [], []
我的系统是 Win2000 ，数据库是 Oracle 8.1.6。
```

老网虫白鳝 12:19:12

用 ANALYZE TABLE ... VALIDATE STRUCTURE 语句分析一下表的结构。

风 12:21:21

我已经分析过了，没有发现错误。

老网虫白鳝 12:22:34

你做一个 10 级的 10046 TRACE，然后执行这个语句，查看报错信息。从 ORA-600 [12700]错误的参数来看，应该是 25 号对象存在问题 ，注意，应该是 DATAOBJ#=25，而不是 OBJ# 。

风 12:22:58

DATAOBJ#25 对应的表是 sys.con$。

老网虫白鳝 12:23:12

如果是表 SYS.CON$，那就对了，这就能解释为什么删除主键时会报错了，看样子不是数据表的问题，而是 CON$的问题。要删除主键，必须要操作 CON$这张表。
接下来，你做一个 10046 TRACE，就可以看到相关的参数了 。
直接执行这个语句是不是也报错?

风 12:26:34

对，出错了
```
ora-600  12700
select * from con$ where name='PK_W_AVG'正常。
select * from con$ where name='PK_W_AVG' and owner#=49 就出错了。
```

老网虫白鳝 12:30:34

CON$表有问题，你加上 FULL 提示看看，应该就不会报错了，两个 SQL 条件不同，走的执行计划也不同，可能一个走了索引一个没走。全表扫描的就没有报错。

风 12:33:19

加 FULL 提示，是可以执行的，没有报错。

老网虫白鳝 12:34:10

看样子是索引坏了。先验证一下索引吧 。

风 12:41:12

```
SQL> ANALYZE index I_CON1 VALIDATE STRUCTURE;索引已分析
SQL> ANALYZE index I_CON2 VALIDATE STRUCTURE;索引已分析
SQL> alter table w_avg drop primary key;
alter table w_avg drop primary key
*
ERROR at line 1:
ORA-00600: internal error code, arguments: [12700], [25], [4228580], [178], [],
[], [], []
```
删除主键时是换了一个用户的，前面是用 SYS 执行的。但换了用户还是一样报错。

老网虫白鳝 12:43:46

通过 OWNER 加上 NAME 的条件访问 CON$使用的索引是 I_CON1，我发一个 oerr12700.sql 给你，这里有两个存储过程，执行一下，看看结果。
这是一个分析脚本，要用 SYS 账号创建存储过程。

风 12:51:46

```
SQL> execute oerr12700(25,4228580,178);
ORA-600 [12700] [25],[4228580],[178]
--------------------------------------------------
there is an index pointing to a row in SYS.CON$
row is slot 178 in file 1 block 34276
one index entry is pointing to ROWID='AAAAAZAABAAAIXkACy'
--------------------------------------------------
You may want to check the integrity of SYS.CON$
executing :
dbv file=C:\ORACLE\ORADATA\GDLS2003\SYSTEM01.DBF
blocksize=8192 start=34276
end=34276
--------------------------------------------------
IF dbv does not show any corruption, you can try to
find the corrupted indexes using the queries proposed
by the procedure oerr12700diag(25,4228580,178)
-------------------------------------------------------
PL/SQL procedure successfully completed.
```

风 12:52:16

```
SQL> execute oerr12700diag(25,4228580,178);
--------------------------------------------------
IF dbv did not show any corruption, you can try to
find the corrupted indexes using following queries:
-------------------------------------------------------
If a query returns "no rows selected" index is sane
If a query returns AAAAAZAABAAAIXkACy index is corrupted
..................................................
.
To test  SYS.CON$ indexes
.
.
To test  INDEX I_CON1 you run :
.
select rowid "I_CON1 corrupted!"
from
(SELECT /*+ INDEX_FFS(CON$,I_CON1) */
OWNER#,rowid from SYS.CON$ where OWNER#=OWNER#)
where rowid='AAAAAZAABAAAIXkACy';
.
To test  INDEX I_CON2 you run :
.
select rowid "I_CON2 corrupted!"
from
(SELECT /*+ INDEX_FFS(CON$,I_CON2) */
CON#,rowid from SYS.CON$ where CON#=CON#)
where rowid='AAAAAZAABAAAIXkACy';

PL/SQL procedure successfully completed.
```

老网虫白鳝 12:53:06

```
select rowid "I_CON1 corrupted!"from
(SELECT /*+ INDEX_FFS(CON$,I_CON1) */
  OWNER#,rowid from SYS.CON$
  where OWNER#=OWNER#)where rowid='AAAAAZAABAAAIXkACy';
```

执行一下上面的那个查询语句，看看是什么结果。

风 12:53:53

```
SQL> select rowid "I_CON1 corrupted!"
2  from
3  (SELECT /*+ INDEX_FFS(CON$,I_CON1) */
4  OWNER#,rowid from SYS.CON$ where OWNER#=OWNER#)
5  where rowid='AAAAAZAABAAAIXkACy';
I_CON1 corrupted!------------------AAAAAZAABAAAIXkACy
```

老网虫白鳝 12:54:47

可以用 DBMS_ROWID 包的函数查看块号、文件号和序号，这样就找到索引损坏的位置了，然后可以使用 BBED 工具去修改。

I_CON1 是一个 BOOTSTRAP$对象，不能重建，只能通过 BBED 工具修改。如果是普通的索引就好办了，直接重建即可。

风 12:56:17

对，不能重建，也不能删除。

老网虫白鳝 12:56:54

我以前碰到过类似的问题，出现 ORA-8102 错误，后来也是用 BBED 工具修改解决的。

风 12:59:34

```
DATA_OBJECT_ID#     RFILE#     BLOCK#       ROW#
--------------- ---------- ---------- ----------
             25          1      34276        178
```

这个是那个 ROWID 的内容。

老网虫白鳝 13:02:57

把 1/34276 块转储出来。

风 13:18:28

已经转储出来了

如何判断是否跟前面的 ROWID 一样?

老网虫白鳝 13:21:26

用 FULL 提示取出 ROWID，AAAAAZAABAAAIXkABd，和刚才的不一样吧?

风 13:26:23

不一样，前面是 AAAAAZAABAAAIXkACy。

```
DATA_OBJECT_ID#     RFILE#     BLOCK#       ROW#
--------------- ---------- ---------- ----------
             25          1      34276        178
```

老网虫白鳝 13:27:04

从块转储中可以看到，ROWID 对应的记录是 0x176:pri[178] sfll=-1。

这表示该记录不存在，sfll 应该指向一个字节偏移量，如果是“－1”就说明这条记录可能已经删除了。

风 13:27:44

哦，就是少了这行，那如何去操作呢?

老网虫白鳝 13:28:11

要将索引中的 ROWID 修改为真实的 ROWID ,只能使用 BBED 工具。首先通过索引找到那条记录,然后通过 select /*+ full(a) */ rowid,... From con$ a where owner=... And name =....语句找到 ROWID，再用 DBMS_ROWID 包解析出 FILE#、BLOCK#和 ROW#，重新组合为索引项中的 ROWID，最后用 BBED 工具将索引项修改为这个值，就可以了。
不过修改前一定要备份数据文件，一旦出现问题还可以恢复。

老网虫白鳝 13:36:52

这个操作对你来说可能有点难度，要了解数据块的结构才能做到。最简单的方法还是导出数据，然后重建数据库。

风 13:40:37

想试验一下，但是不知道怎么操作。
唉，刚才和应用那边沟通过了，没时间了。只能导出数据重建数据库了。
呵呵，非常感谢老白。

由于“风”对手工修改数据字典并无把握，因此这个问题并没有深入讨论下去，后来老白通过一个实验重现了这个故障，并记录了故障处理的全部过程。

```
SQL> drop index idx_t1;
drop index idx_t1
           *
ERROR at line 1:
ORA-00600: internal error code, arguments: [ktssdrp1], [4], [4], [130707], [],
[], [], []
```

在 ALERT LOG 日志中，可以找到如下信息：

```
Mon Feb 13 11:03:50 2012
Errors in file /opt/oracle/admin/orcl/udump/orcl_ora_26205.trc:
ORA-00600: internal error code, arguments: [ktssdrp1], [4], [4], [130707], [], [], [],
[]
```

打开 orcl_ora_26205.trc 文件，可以看到：

```
*** 2012-02-13 11:03:50.714
ksedmp: internal or fatal error
ORA-00600: internal error code, arguments: [ktssdrp1], [4], [4], [130707], [], [], [],
[]
Current SQL statement for this session:
drop index idx_t1
----- Call Stack Trace -----
calling              call     entry                argument values in hex
location             type     point                (? means dubious value)
-------------------- -------- -------------------- ----------------------------
ksedst()+27          call     ksedst1()            0 ? 1 ?
ksedmp()+557         call     ksedst()             0 ? 0 ? 0 ? 0 ? 67A035F0 ?
                                                   0 ?
ksfdmp()+19          call     ksedmp()             3 ? BFFFB2F8 ? ADC338D ?
```

```
                                                           CE67960 ? 3 ? CE15B3C ?
kgerinv()+177        call     00000000                     CE67960 ? 3 ?
kgeasnmierr()+40     call     kgerinv()                    CE67960 ? CED2C28 ? C393C24 ?
                                                           3 ? BFFFB330 ?
ktssdrp_segment()+1  call     kgeasnmierr()                CE67960 ? CED2C28 ? C393C24 ?
749                                                        3 ? 0 ? 4 ?
dixdrv()+2540        call     ktssdrp_segment()            BFFFB578 ? 0 ? B727E7C0 ?
                                                           41 ? 1 ? BFFFB598 ?
opiexe()+11140       call     dixdrv()                     A ? B727FBBC ? 0 ? C ?
                                                           ADA884D ? B727FBBC ?
opiosq0()+2701       call     opiexe()                     4 ? 0 ? BFFFC1B0 ?
kpooprx()+215        call     opiosq0()                    3 ? E ? BFFFC2AC ? A4 ?
kpoal8()+673         call     kpooprx()                    BFFFED94 ? BFFFD024 ? 11 ?
                                                           1 ? 0 ? A4 ?
opiodr()+976         call     00000000                     5E ? 17 ? BFFFED90 ?
ttcpip()+1085        call     00000000                     5E ? 17 ? BFFFED90 ? 0 ?
opitsk()+1054        call     ttcpip()                     CE6F180 ? 5E ? BFFFED90 ? 0 ?
                                                           BFFFEA70 ? BFFFEEA0 ?
opiino()+821         call     opitsk()                     0 ? 0 ?
opiodr()+976         call     00000000                     3C ? 4 ? BFFFF960 ?
opidrv()+466         call     opiodr()                     3C ? 4 ? BFFFF960 ? 0 ?
sou2o()+91           call     opidrv()                     3C ? 4 ? BFFFF960 ?
opimai_real()+117    call     sou2o()                      BFFFF944 ? 3C ? 4 ?
                                                           BFFFF960 ?
main()+111           call     opimai_real()                2 ? BFFFF990 ?
__libc_start_main()  call     00000000                     2 ? BFFFFA54 ? BFFFFA60 ?
+211                                                       925056 ? A5AFF4 ? 0 ?

--------------------- Binary Stack Dump ---------------------
```

在 trace 文件中并没有看到当前执行的 SQL，所以需要进行一次 10046 TRACE，这里只需分析问题是出现在哪条 SQL 上的，因此不需要很高级别的跟踪，一般的 SQL_TRACE 就能满足要求了。

```
SQL> conn /as sysdba
Connected.
SQL> alter session set sql_trace=true;

Session altered.

SQL> drop index scott.idx_t1;
drop index scott.idx_t1
                 *
ERROR at line 1:
ORA-00600: internal error code, arguments: [ktssdrp1], [4], [4], [130707], [],
[], [], []
```

trace 文件已经产生了，下一步要打开 trace 文件来检查到底在哪条 SQL 上出现了问题：

```
STAT #6 id=1 cnt=0 pid=0 pos=1 obj=14 op='TABLE ACCESS CLUSTER SEG$ (cr=3 pr=0 pw=0
time=56 us)'
STAT #6 id=2 cnt=1 pid=1 pos=1 obj=9 op='INDEX UNIQUE SCAN I_FILE#_BLOCK# (cr=2 pr=0
pw=0 time=31 us)'
*** 2012-02-13 11:24:50.213
```

```
ksedmp: internal or fatal error
ORA-00600: internal error code, arguments: [ktssdrp1], [4], [4], [130707], [], [], [],
[]
Current SQL statement for this session:
drop index scott.idx_t1
```

我们可以看到，在访问 SEG$的时出现了问题，这种情况很可能是数据字典不一致引起的。Oracle 有一个工具包 hcheck，通过该工具包可以检查数据字典不一致带来的问题。由于这个系统没有安装过 hcheck，我们首先要安装 hcheck 工具。目前 hcheck 的版本有 2.0 和 3.5 两个，3.5 版本适用于 9*i* 及更新的版本，2.0 版本适用于 8*i* 及更早的版本。要使用 hcheck 工具必须先安装 hout.sql（参见 Metalink 101468.1）：

```
[oracle@justdb ~]$ sqlplus '/as sysdba'

SQL*Plus: Release 10.2.0.4.0 - Production on Mon Feb 13 14:57:22 2012

Copyright (c) 1982, 2007, Oracle.  All Rights Reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.4.0 - Production
With the Partitioning, Oracle Label Security, OLAP, Data Mining
and Real Application Testing options

SQL> @hout

Package created.

No errors.

Package body created.

SQL> @hcheck3.sql

Package created.

Package body created.

H.Check Version 9i+/hc3.35
---------------------------------------
Catalog Version 10.2.0.4.0 (1002000400)
---------------------------------------

                                   Catalog       Fixed
Procedure Name                     Version    Vs Release      Run
------------------------------ ... ---------- -- ----------   ---
.- SynLastDDLTim               ... 1002000400 >  1001000200 : n/a
.- LobNotInObj                 ... 1002000400 >  1000000200 : n/a
.- MissingOIDOnObjCol          ... 1002000400 <=  *All Rel* : Ok
.- SourceNotInObj              ... 1002000400 >  1002000100 : n/a
.- IndIndparMismatch           ... 1002000400 <= 1102000100 : Ok
.- InvCorrAudit                ... 1002000400 <= 1102000100 : Ok
.- OversizedFiles              ... 1002000400 <=  *All Rel* : Ok
.- TinyFiles                   ... 1002000400 >   900010000 : n/a
```

```
.- PoorDefaultStorage          ... 1002000400 <=  *All Rel* : Ok
.- PoorStorage                 ... 1002000400 <=  *All Rel* : Ok
.- MissTabSubPart              ... 1002000400 >   900010000 : n/a
.- PartSubPartMismatch         ... 1002000400 <= 1102000100 : Ok
.- TabPartCountMismatch        ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTabComPart          ... 1002000400 >   900010000 : n/a
.- ZeroTabSubPart              ... 1002000400 >   902000100 : n/a
.- MissingSum$                 ... 1002000400 <=  *All Rel* : Ok
.- MissingDir$                 ... 1002000400 <=  *All Rel* : Ok
.- DuplicateDataobj            ... 1002000400 <=  *All Rel* : Ok
.- ObjSynMissing               ... 1002000400 <=  *All Rel* : Ok
.- ObjSeqMissing               ... 1002000400 <=  *All Rel* : Ok
.- OrphanedUndo                ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndex               ... 1002000400 <=  *All Rel* : Ok

HCKE-0016: Orphaned IND$ (no SEG$)
ORPHAN IND$: OBJ=64917 DOBJ=64917 TS=4 RFILE/BLOCK=4 130707 BO#=64916 SegType=

.- OrphanedIndexPartition      ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndexSubPartition   ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTable               ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTablePartition      ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTableSubPartition   ... 1002000400 <=  *All Rel* : Ok
.- MissingPartCol              ... 1002000400 <=  *All Rel* : Ok
.- OrphanedSeg$                ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndPartObj#         ... 1002000400 <= 1101000600 : Ok
.- DuplicateBlockUse           ... 1002000400 <=  *All Rel* : Ok
.- HighObjectIds               ... 1002000400 >   801060000 : n/a
.- PQsequence                  ... 1002000400 >   800060000 : n/a
.- TruncatedCluster            ... 1002000400 >   801070000 : n/a
.- FetUet                      ... 1002000400 <=  *All Rel* : Ok
.- Uet0Check                   ... 1002000400 <=  *All Rel* : Ok
.- ExtentlessSeg               ... 1002000400 <=  *All Rel* : Ok
.- SeglessUET                  ... 1002000400 <=  *All Rel* : Ok
.- BadInd$                     ... 1002000400 <=  *All Rel* : Ok
.- BadTab$                     ... 1002000400 <=  *All Rel* : Ok
.- BadIcolDepCnt               ... 1002000400 <= 1101000700 : Ok
.- WarnIcolDep                 ... 1002000400 <= 1101000700 : Ok
.- OnlineRebuild$              ... 1002000400 <=  *All Rel* : Ok
.- DropForceType               ... 1002000400 >  1001000200 : n/a
.- TrgAfterUpgrade             ... 1002000400 <=  *All Rel* : Ok
.- FailedInitJVMRun            ... 1002000400 <=  *All Rel* : Ok
.- TypeReusedAfterDrop         ... 1002000400 >   900010000 : n/a
.- Idgen1$TTS                  ... 1002000400 >   900010000 : n/a
.- DroppedFuncIdx              ... 1002000400 >   902000100 : n/a
.- BadOwner                    ... 1002000400 >   900010000 : n/a
.- UpgCheckc0801070            ... 1002000400 <=  *All Rel* : Ok
.- BadPublicObjects            ... 1002000400 <=  *All Rel* : Ok
.- BadSegFreelist              ... 1002000400 <=  *All Rel* : Ok
.- BadCol#                     ... 1002000400 >  1001000200 : n/a
.- BadDepends                  ... 1002000400 <=  *All Rel* : Ok
.- CheckDual                   ... 1002000400 <=  *All Rel* : Ok
.- ObjectNames                 ... 1002000400 <=  *All Rel* : Ok
.- BadCboHiLo                  ... 1002000400 <=  *All Rel* : Ok
.- ChkIotTs                    ... 1002000400 <=  *All Rel* : Ok
```

```
.- NoSegmentIndex            ... 1002000400 <=  *All Rel* : Ok
.- BadNextObject             ... 1002000400 <=  *All Rel* : Ok
.- OrphanIndopt              ... 1002000400 >   902000800 : n/a
.- UpgFlgBitTmp              ... 1002000400 >  1001000100 : n/a
.- RenCharView               ... 1002000400 >  1001000100 : n/a
.- Upg9iTab$                 ... 1002000400 >   902000400 : n/a
.- Upg9iTsInd                ... 1002000400 >   902000500 : n/a
.- Upg10gInd$                ... 1002000400 >  1002000000 : n/a
.- DroppedROTS               ... 1002000400 <=  *All Rel* : Ok
.- ChrLenSmtcs               ... 1002000400 <= 1101000600 : Ok
.- FilBlkZero                ... 1002000400 <=  *All Rel* : Ok

Found 1 potential problem(s) and 0 warning(s)
Contact Oracle Support with the output
to check if the above needs attention or not

PL/SQL procedure successfully completed.

SQL>
```

我们可以看到，在本次扫描中出现了一个错误：

```
HCKE-0016: Orphaned IND$ (no SEG$)
ORPHAN IND$: OBJ=64917 DOBJ=64917 TS=4 RFILE/BLOCK=4 130707 BO#=64916 SegType=
```

这个错误说明 IND$中的对象没有 SEG$信息，OBJECT ID 是 64917，我们可以验证一下：

```
SQL> col object_name format a30 trunc
SQL> set line 132
SQL>  select owner,object_name from dba_objects where object_id=64917;

OWNER                          OBJECT_NAME
------------------------------ ------------------------------
SCOTT                          IDX_T1
```

确实是要删除的这个索引。从几方面的分析来看，都定位到了这个索引，因此下一步我们就需要修复这个问题，在 SEG$中手工输入相关的数据。由于修改字典数据是十分危险的操作，因此建议操作之前对数据库进行备份。另外，进行该操作时最好停掉业务系统，在维护模式下操作。

首先关闭数据库，然后重新以限制模式启动数据库：

```
SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE Instance shut down.
SQL> startup restrict;
ORACLE Instance started.

Total System Global Area 1241513984 bytes
Fixed Size                  1267212 bytes
Variable Size             436210164 bytes
Database Buffers          788529152 bytes
Redo Buffers               15507456 bytes
Database mounted.
Database opened.
```

```
SQL>  alter system set "_smu_debug_mode"=1  scope=memory;

System altered.

SQL> set transaction use rollback segment system;

Transaction set.

SQL> select user_id,username from dba_users where username='SCOTT';

   USER_ID USERNAME
---------- ------------------------------
        57 SCOTT

SQL> select count(*) from dba_extents where segment_name='IDX_T1' and owner='SCOTT';

  COUNT(*)
----------
        16
```

在正式插入数据前，还需要知道这张表的 USER#信息，以及扩展数量信息，有了这些信息，就可以执行下面的 INSERT 语句，手工修改 SEG$：

```
SQL> insert into seg$
  2  (
  3   file#, block#, type#, ts#, blocks, extents,
  4   iniexts, minexts, maxexts,extsize, extpct, user#,
 bitmapranges, cachehint, scanhint, hwmincr
)
values
  (
  4, 130707, 6, 4, 1, 1,
  5    6    7    8    9   10    8, 1, 2147483645,128, 0, 57,
 11    0, 0, 0, 1
); 12

1 row created.

SQL> commit;

Commit complete.
```

完成操作后，可以重启数据库，然后再次删除索引：

```
SQL> drop index idx_t1;
drop index idx_t1
           *
ERROR at line 1:
ORA-00603: ORACLE server session terminated by fatal error
```

大家可能会因为这个问题惊出一身冷汗，因为查看 ALERT LOG 日志时，会发现：

```
Mon Feb 13 15:56:36 2012
Errors in file /opt/oracle/admin/orcl/bdump/orcl_smon_24807.trc:
ORA-00600: internal error code, arguments: [ktssdro_segment1], [4], [16867091], [0],
```

```
[], [], [], []
Mon Feb 13 15:56:38 2012
Errors in file /opt/oracle/admin/orcl/bdump/orcl_pmon_24794.trc:
ORA-00474: SMON process terminated with error
Mon Feb 13 15:56:38 2012
PMON: terminating Instance due to error 474
Instance terminated by PMON, pid = 24794
```

由于出现了 ORA-600 [ktssdro_segment1]错误，数据库实例宕机了。不要害怕，这是因为第一次删除索引时，已经有一部分操作导致数据字典和数据文件出现了不一致，因此在第二次删除索引时，SEG$的数据没有正常删除。我们需要删除 SEG$的数据，根据刚才插入 SEG$的语法，编写一个 DELETE 命令，必须在数据库重新启动后马上执行删除操作，否则数据库实例还是会宕机。

```
[oracle@justdb ~]$ sqlplus '/as sysdba'

SQL*Plus: Release 10.2.0.4.0 - Production on Mon Feb 13 16:13:05 2012

Copyright (c) 1982, 2007, Oracle.  All Rights Reserved.

Connected to an idle Instance.

SQL> startup restrict;
ORACLE Instance started.

Total System Global Area 1241513984 bytes
Fixed Size                  1267212 bytes
Variable Size             436210164 bytes
Database Buffers          788529152 bytes
Redo Buffers               15507456 bytes
Database mounted.
Database opened.
SQL> delete from seg$ where file#=4 and block#=130707 and ts#=4;

1 row deleted.

SQL> commit;

Commit complete.

SQL> shutdown abort
ORACLE Instance shut down.
SQL> startup
ORACLE Instance started.

Total System Global Area 1241513984 bytes
Fixed Size                  1267212 bytes
Variable Size             436210164 bytes
Database Buffers          788529152 bytes
Redo Buffers               15507456 bytes
Database mounted.
Database opened.
```

数据库正常打开了，而且 ALERT LOG 日志没有再次报错，我们用 hcheck 工具再次检查数据字典。

```
SQL> set serveroutput on size unlimited
SQL> execute hcheck.full;
H.Check Version 9i+/hc3.35
---------------------------------------
Catalog Version 10.2.0.4.0 (1002000400)
---------------------------------------

                                   Catalog       Fixed
Procedure Name                     Version    Vs Release      Run
------------------------------ ... ---------- -- ----------   ---
.- SynLastDDLTim               ... 1002000400 >  1001000200 : n/a
.- LobNotInObj                 ... 1002000400 >  1000000200 : n/a
.- MissingOIDOnObjCol          ... 1002000400 <=  *All Rel* : Ok
.- SourceNotInObj              ... 1002000400 >  1002000100 : n/a
.- IndIndparMismatch           ... 1002000400 <= 1102000100 : Ok
.- InvCorrAudit                ... 1002000400 <= 1102000100 : Ok
.- OversizedFiles              ... 1002000400 <=  *All Rel* : Ok
.- TinyFiles                   ... 1002000400 >   900010000 : n/a
.- PoorDefaultStorage          ... 1002000400 <=  *All Rel* : Ok
.- PoorStorage                 ... 1002000400 <=  *All Rel* : Ok
.- MissTabSubPart              ... 1002000400 >   900010000 : n/a
.- PartSubPartMismatch         ... 1002000400 <= 1102000100 : Ok
.- TabPartCountMismatch        ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTabComPart          ... 1002000400 >   900010000 : n/a
.- ZeroTabSubPart              ... 1002000400 >   902000100 : n/a
.- MissingSum$                 ... 1002000400 <=  *All Rel* : Ok
.- MissingDir$                 ... 1002000400 <=  *All Rel* : Ok
.- DuplicateDataobj            ... 1002000400 <=  *All Rel* : Ok
.- ObjSynMissing               ... 1002000400 <=  *All Rel* : Ok
.- ObjSeqMissing               ... 1002000400 <=  *All Rel* : Ok
.- OrphanedUndo                ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndex               ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndexPartition      ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndexSubPartition   ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTable               ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTablePartition      ... 1002000400 <=  *All Rel* : Ok
.- OrphanedTableSubPartition   ... 1002000400 <=  *All Rel* : Ok
.- MissingPartCol              ... 1002000400 <=  *All Rel* : Ok
.- OrphanedSeg$                ... 1002000400 <=  *All Rel* : Ok
.- OrphanedIndPartObj#         ... 1002000400 <= 1101000600 : Ok
.- DuplicateBlockUse           ... 1002000400 <=  *All Rel* : Ok
.- HighObjectIds               ... 1002000400 >   801060000 : n/a
.- PQsequence                  ... 1002000400 >   800060000 : n/a
.- TruncatedCluster            ... 1002000400 >   801070000 : n/a
.- FetUet                      ... 1002000400 <=  *All Rel* : Ok
.- Uet0Check                   ... 1002000400 <=  *All Rel* : Ok
.- ExtentlessSeg               ... 1002000400 <=  *All Rel* : Ok
.- SeglessUET                  ... 1002000400 <=  *All Rel* : Ok
.- BadInd$                     ... 1002000400 <=  *All Rel* : Ok
.- BadTab$                     ... 1002000400 <=  *All Rel* : Ok
.- BadIcolDepCnt               ... 1002000400 <= 1101000700 : Ok
```

```
.- WarnIcolDep                 ... 1002000400 <= 1101000700 : Ok
.- OnlineRebuild$              ... 1002000400 <=  *All Rel* : Ok
.- DropForceType               ... 1002000400 >  1001000200 : n/a
.- TrgAfterUpgrade             ... 1002000400 <=  *All Rel* : Ok
.- FailedInitJVMRun            ... 1002000400 <=  *All Rel* : Ok
.- TypeReusedAfterDrop         ... 1002000400 >   900010000 : n/a
.- Idgen1$TTS                  ... 1002000400 >   900010000 : n/a
.- DroppedFuncIdx              ... 1002000400 >   902000100 : n/a
.- BadOwner                    ... 1002000400 >   900010000 : n/a
.- UpgCheckc0801070            ... 1002000400 <=  *All Rel* : Ok
.- BadPublicObjects            ... 1002000400 <=  *All Rel* : Ok
.- BadSegFreelist              ... 1002000400 <=  *All Rel* : Ok
.- BadCol#                     ... 1002000400 >  1001000200 : n/a
.- BadDepends                  ... 1002000400 <=  *All Rel* : Ok
.- CheckDual                   ... 1002000400 <=  *All Rel* : Ok
.- ObjectNames                 ... 1002000400 <=  *All Rel* : Ok
.- BadCboHiLo                  ... 1002000400 <=  *All Rel* : Ok
.- ChkIotTs                    ... 1002000400 <=  *All Rel* : Ok
.- NoSegmentIndex              ... 1002000400 <=  *All Rel* : Ok
.- BadNextObject               ... 1002000400 <=  *All Rel* : Ok
.- OrphanIndopt                ... 1002000400 >   902000800 : n/a
.- UpgFlgBitTmp                ... 1002000400 >  1001000100 : n/a
.- RenCharView                 ... 1002000400 >  1001000100 : n/a
.- Upg9iTab$                   ... 1002000400 >   902000400 : n/a
.- Upg9iTsInd                  ... 1002000400 >   902000500 : n/a
.- Upg10gInd$                  ... 1002000400 >  1002000000 : n/a
.- DroppedROTS                 ... 1002000400 <=  *All Rel* : Ok
.- ChrLenSmtcs                 ... 1002000400 <= 1101000600 : Ok
.- FilBlkZero                  ... 1002000400 <=  *All Rel* : Ok

Found 0 potential problem(s) and 0 warning(s)

PL/SQL procedure successfully completed.
```

可以看到，hcheck 工具没有发现问题，数据字典的问题已经解决了，故障也排除了。不过大家可能会认为上面的处理过程太过惊险。实际上，对于这个 ORA-600 错误，这种处理方法是肯定可以解决问题的，不过对于其他的错误，我们并不清楚修改数据字典可能会带来怎样的影响，因此在没有经过严格测试或者对数据字典并不十分了解的情况下，不建议进行修改数据字典的操作。因为一旦出现问题，可能导致数据库出现更大的故障。

17.2　ORA-600 [kdsgrp1]的处理案例

我的一个客户最近碰到了点烦心的事情，他在一套数据库上面做了导出备份，想用移动硬盘把备份的数据复制出来，可没想到一连上移动硬盘，整个服务器就掉电了。重新开机后，数据库倒是能够打开，不过有些应用模块却出错了，查看 ALERT LOG 日志后，发现存在大量的 ORA-600 [kdsgrp1]错误。

这种问题我以前并没有碰到过，于是就到 Metalink 上去查找，查询结果如图 17-1 所示。

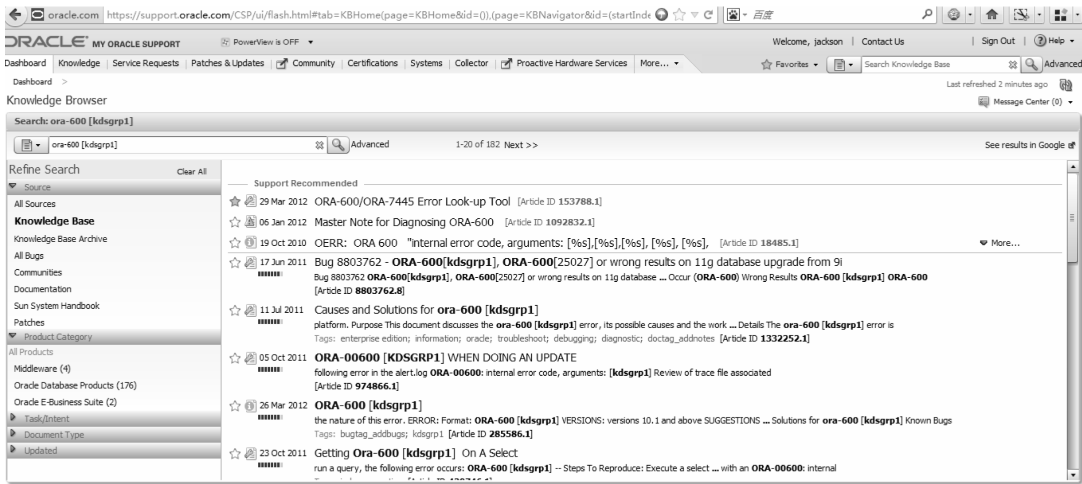


图　17-1

从查询到的文档标题可以看出，Causes and Solutions for ora-600 [kdsgrp1] [ID 1332252.1]像是能够给我们提供答案的文档。这份文档的第一部分说明了该文档适用的 Oracle 版本和环境。

```
Applies to:
Oracle Server - Enterprise Edition - Version: 10.2.0.4 to 11.2.0.2 - Release: 10.2 to
11.2
Information in this document applies to any platform.
```

上述内容表示该文档适用于 10.2 ~ 11.2 版本，具体的版本号是 10.2.0.4 ~ 11.2.0.2，如果这个版本范围和我们出现问题的系统的版本相差较大，那么文档内容和我们碰到的问题的关联性也就不高。因此这份资料仅仅能起到参考作用，我们不能完全照搬其解决方法。

```
Purpose
This document discusses the ora-600 [kdsgrp1] error, its possible causes and the work
around solutions that can be tried.
```

这部分内容阐明了该文档的目的，针对本案例，介绍了这个 ORA-600 错误的产生原因以及解决方法。

```
Last Review Date
June 22, 2011
```

上述内容是最近被重新评审的时间，如果已经有好多年没有经过评审了，那么这份文档中信息的准确性和时效性就要大打折扣了。

```
Instructions for the Reader
A Troubleshooting Guide is provided to assist in debugging a specific issue. When
possible, diagnostic tools are included in the document to assist in troubleshooting.
```

这部分内容是对读者的建议。再往后就是故障分析的详细方法了。

```
Troubleshooting Details
The ora-600 [kdsgrp1] error is thrown when a fetch operation fails to find the expected
```

```
row. The error is hit in memory and so may be a memory only error or an error that results
from corruption on disk.
This error may indicate (but is not restricted to) any of the following conditions:
?   Lost writes
?   Parallel DML issues
?   Index corruption
?   Data block corruption
?   Consistent read [CR] issues
?   Buffer cache corruption
A full list of known issues is given in ORA-600 [kdsgrp1] Note 285586.1 Each bug has
a short description that indicates the circumstances where it is hit. The bug list can
be shortened by selecting your database release to show only those issues that may affect
you.
This issue may be intermittent or it may persist until the underlying disk level
corruption is fixed. Intermittent issues are likely to be memory based (however
intermittent access to the corruption can be confused with intermittent memory issues).
Common Work Around Solutions
If the issue is in memory only we can try to immediately resolve the issue by flushing
the buffer cache but remember to consider the performance impact on production systems:
alter system flush buffer_cache;
If we have an intermittent consistent read issue we can try disabling rowCR which is
an optimization to reduce consistent-read rollbacks during queries by setting_
row_cr=FALSE in the initialization files. However, this could lead to performance
degradation of queries. Please check the ratio of the two statistics "RowCR hits"/"RowCR
attempts" to determine whether the workaround is to be used.
If this is a result of index corruption then we can drop and rebuild the index. Note
that this will require a maintenance window on production systems.
Root Cause Determination
Now lets look at how we discover the root cause of the problem: the first step in finding
the root cause of this issue is to inspect the generated trace file. The ora-600 will
generate both a trace file in the trace directory and an incident file under the incident
id within the incident directory.
The top part of the trace file tells us the SQL that was being run when the error was
hit:
----- Current SQL Statement for this session (sql_id=9mamr7xn4wg7x) -----
This immediately shows us the data objects that were accessed. Searching the trace file
for the text string 'Plan Table' will locate the SQL execution plan that is dumped within
this trace file. For a persistent issue this allows us to determine which indexes have
been accessed and so identify indexes that should be validated to check for block
corruption:
SQL> analyze index scott.pk_dept validate structure online;
 Index analyzed.
An other approach we can take is to use the file and block information contained in
the trace file. At the top of the trace file we will find information on the block where
the corruption was found:
*** SESSION ID:(3202.5644) 2011-03-19 04:12:16.910
 row 07c7c8c7.a continuation at
 file# 31 block# 510151 slot 11 not found
This information can be used to identify the object details in dba_extents:
Select owner, segment_name, segment_type, partition_name,tablespace_name
 From dba_extents
 Where relative_fno = <file id>
 And <block#> between block_id and (block_id+blocks-1);
We can then validate this object, for example a table and all it's indexes:
```

```
Analyze table scott.dept validate structure cascade online;
Remember that we may be dealing with a permanent corruption that is not located in the
object blocks themselves. Examples of this include:
?   Dictionary corruption issue from transportable tablespace operations: check
dba_tablespaces to see if the tablespace has been plugged in.
?   Lost writes in ASM diskgroup mirrors - most likely to be seen when there is heavy
IO and disk resync activity. To check this run dbms_diskgroup.checkfile to detect mirror
discrepancies
If analyze reports no corruption then check if there are any chained rows on the table.
If these exist then we may have an undetected corruption and the issue should reproduce
whenever the SQL is run. Exporting the table will also detect this issue.
If analyze and exporting the table (in the presence of chained rows) both report no
errors then this should be considered a consistent read issue.
Once you understand the nature of the problem you can review the list of known bugs
and determine which one matches your condition. If you cannot determine which issue
is affecting you then open a service request with Oracle Support and upload the RDBMS
and ASM (if applicable)Instance alert logs for all nodes, any trace and incident files
generated and a full description of the nature of the problem.
```

通过对上面资料的分析，再对照 ORA-600 的 trace 文件，我们发现系统的故障和这份文档所描述的情况具有很大的相似性。首先我们需要排除由于内存故障导致的问题。

```
ALTER SYSTEM FLUSH BUFFER_POOL;
```

执行上面的语句后，问题并没有消失，打开 trace 文件，可以看到以下信息：

```
*** SESSION ID:(886.204) 2012-04-23 09:53:56.577
            row 0f924e83.15 continuation at
            file# 62 block# 1199747 slot 22 not found
**************************************************
KDSTABN_GET: 0 ..... ntab: 1
curSlot: 22 ..... nrows: 14
**************************************************
```

不难看出，FILE 62、BLOCK 1199747 的 SLOT 22 存在问题，通过以下查询，可以找到相关的对象。

```
select owner,segment_name from dba_extents where file_id=62 and block_id<=1199747 and
(block_id+blocks)>=1199747
```

在同一个 trace 文件中继续查找，可以找到相关 SQL 及执行计划：

```
============
Plan Table
============
-----------------------------------------------------------------+-----------------+
| Id | Operation                         | Name                    | Rows | Time     |
-----------------------------------------------------------------+-----------------+
| 0  | SELECT STATEMENT                  |                         |      |          |
| 1  | COUNT STOPKEY                     |                         |      |          |
| 2  | NESTED LOOPS OUTER                |                         |   2  | 00:00:01 |
| 3  | VIEW                              |                         |   2  | 00:00:01 |
| 4  | SORT ORDER BY                     |                         |   2  | 00:00:01 |
| 5  | TABLE ACCESS BY INDEX ROWID       |CHOUJIANG                |   2  | 00:00:01 |
| 6  | INDEX RANGE SCAN                  |INDEX_CHOUJIANG_MOBILENO|   3  | 00:00:01 |
```

```
| 7  | TABLE ACCESS BY INDEX ROWID      | ACTDATA_STAT_LOG        |   1   | 00:00:01 |
| 8  | INDEX UNIQUE SCAN                | PK_ACTSTAT_BATCHID      |   1   |          |
-----------------------------------------------------------------+-----------------+
```

我们发现，CHOUJIANG 表访问使用了 INDEX_CHOUJIANG_MOBILIENO 这个索引。那么下一步要做的就是删除该索引，然后进行重建（需要重建该索引，而不是 ALTER INDEX ... REBUILD）。重建索引后这个错误就应该消失了。

由于晚上只有部分业务在使用，因此还无法排除其他索引也存在类似的问题。于是我们找出系统中业务相关的所有表，通过以下命令编写了一个脚本，对主要的表和相关索引进行分析。

```
ANALYZE TABLE CHOUJIANG VALIDATE STRUCTURE CASCADE ONLINE;
```

如果分析过程中出现如下所示的 ORA-1499 错误，那么就说明该表的索引存在问题。

```
ERROR at line 1:

ORA-01499: table/index cross reference failure - see trace file
```

如果找到 trace 文件，可以看到 trace 文件中存在如下信息：

```
*** ACTION NAME:() 2012-04-23 10:43:03.440
*** MODULE NAME:(sqlplus@dgser_CRM_M (TNS V1-V3)) 2012-04-23 10:43:03.440
*** SERVICE NAME:(SYS$USERS) 2012-04-23 10:43:03.439
*** SESSION ID:(877.4034) 2012-04-23 10:43:03.439
row not found in index tsn: 17 rdba: 0x3409dcc4
env: (scn: 0x0004.9fffeb07  xid: 0x0000.000.00000000  uba: 0x00000000.0000.00
statement num=0 parent xid: xid: 0x0000.000.00000000 scn: 0x0000.00000000 8sch: scn:
0x0000.00000000)
col 0; len 11; (11):  31 33 37 31 32 36 36 33 32 39 34
col 1; len 6; (6):  04 09 01 55 00 0b
```

这些信息已经充分证明了索引和表数据存在不一致，因此需要重建索引。因为重建操作是通过当前索引加上索引的增量来进行的，因此通过重建或者在线重建操作，都无法彻底消除这种不一致现象，必须先删除索引再进行重建。

一般来说，这种问题都是由存储方面的故障导致的，因此问题并不会孤立出现，往往很多索引也存在类似的不一致现象。而只有访问到了不一致的记录，系统才会报错，因此需要分析系统中所有相关的索引，可以通过以下脚本进行。

```
ANALYZE TABLE <table_name> VALIDATE STRUCTURE CASCADE ONLINE;
```

如果这张表上存在主键，那么就应该检查主键是否存在问题：

```
LOCK TABLE <TABLE_NAME> IN EXCLUSIVE MODE;
SELECT COUNT(*) FROM <TABLE_NAME>;
SELECT /*+ FULL(TAB1) */ COUNT(*) FROM <TABLE_NAME> TAB1;
```

判断这两个查询结果是否一致。如果不一致，就说明主键存在问题，这种情况处理起来会比较麻烦，需要删除主键约束以及相关索引，然后重建主键和索引。如果数据中存在问题，主键相关字段的唯一性也存在问题，那么就更麻烦了，这时需要找到重复的数据，并且手工删除重复的记录。

查找重复主键的方法有两种。

- 在删除主键后重新添加主键时，使用 exception into 子句，这样存在主键重复的记录的 ROWID 就会被记录到异常表（EXCEPTION TABLE）中。
- 通过以下语句来查找重复主键。

```
select <主键字段>,count(*) from <table name> group by <主键字段> having count(*)>1
```

第 18 章

性能问题分析

18

DBA 经常会碰到性能问题，很多时候都会感到束手无策，因此本章可能是很多 DBA 都想认真研读的。实际上，性能问题和普通问题的分析并无太大的区别。我们在分析时，都需要通过自己掌握的知识来逐步缩小分析范围，并采用一系列“组合拳”来解决问题。本章选用的案例都具有一定的代表性，大家在阅读时，不仅要注意老白是如何处理这些案例的，更重要的是要掌握分析、解决性能问题的方法和思路。

18.1　压力测试遇到的问题

朋友的公司为了投标一个项目，需要进行压力测试。这是一套银行的核心业务系统，客户要求综合交易压力测试能够达到每秒 600 个交易，并且系统负载不能超过 95%。系统安装后，他们进行了自测，但无论怎样调整，系统都只能完成每秒不到 200 个交易，而负载只有 20%左右，根本压不上去。于是我查看了相关的 AWR 报告，发现一个十分有趣的现象。

```
Cache Sizes (end)
~~~~~~~~~~~~~~~~~~~
               Buffer Cache:     20,480M       Std Block Size:         8K
           Shared Pool Size:      5,008M           Log Buffer:    32,768K

Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                  Redo size:          1,989,039.62             12,816.67
              Logical reads:            181,739.18              1,171.06
              Block changes:              7,007.77                 45.16
             Physical reads:              2,112.79                 13.61
            Physical writes:                  0.57                  0.00
                 User calls:             12,191.86                 78.56
                     Parses:              8,958.80                 57.73
                Hard parses:                 49.07                  0.32
                      Sorts:                210.70                  1.36
                     Logons:                  0.03                  0.00
                   Executes:             11,725.29                 75.55
               Transactions:                155.19

 % Blocks changed per Read:     3.86    Recursive Call %:     4.33
```

```
Rollback per transaction %:    22.56       Rows per Sort:    139.64

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:    99.96       Redo NoWait %:   100.00
               Buffer Hit %:    98.84    In-memory Sort %:   100.00
              Library Hit %:   107.03        Soft Parse %:    99.45
         Execute to Parse %:    23.59         Latch Hit %:    95.60
Parse CPU to Parse Elapsd %:     2.89     % Non-Parse CPU:    94.08

     Shared Pool Statistics        Begin      End
                                  -------    ------
             Memory Usage %:        9.52     9.57
    % SQL with executions>1:       94.13    94.34
  % Memory for SQL w/exec>1:       89.00    92.95

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                   % Total
Event                              Waits    Time(s)    DB Time     Wait Class
---------------------------- --------- ---------- ---------- ------------
latch: library cache         179,862     3,360       35.73      Concurrency
CPU time                                 1,391       14.79
latch: cache buffers chains   21,376       642        6.83      Concurrency
latch: library cache lock     26,875       561        5.96      Concurrency
latch: library cache pin      30,675       519        5.52      Concurrency
          -------------------------------------------------------------
```

可以看出，库缓存方面的争用所占的比重非常高，查看详细的等待情况如下：

```
Event                          Waits     Timeouts  Total Wait    Avg wait(ms)    Waits
                                                     Time (s)                     /txn
---------------------------- -------- --------- ----------- ------------ ----------
latch: library cache         179,862         0     3,360           19         14.81
latch: cache buffers chains  21,376     21,376       642           30          1.76
latch: library cache lock    26,875          0       561           21          2.21
latch: library cache pin     30,675          0       519           17          2.53
db file sequential read      40,017          0       233            6          3.29
wait list latch free         4,528           0       127           28          0.37
library cache pin            4,561           0       125           27          0.38
kksfbc child completion      2,260       2,258       109           48          0.19
```

库缓存相关的等待事件在 20 毫秒左右，看来是 SQL 解析出现了问题。软解析所占比例达到 99.45%，不过对硬解析方面的影响并不是很大。接下来，需要分析相关的时间模型。

```
Statistic Name                               Time(seconds)   %Total DB Time
---------------------------------------- -------------- ----------------
DB time                                         9,404.41        100.00
sql execute elapsed time                        3,947.37         41.97
parse time elapsed                              2,818.27         29.97
DB CPU                                          1,391.36         14.79
hard parse elapsed time                            23.87          0.25
background elapsed time                            15.84          0.17
failed parse elapsed time                          12.48          0.13
background cpu time                                 5.09          0.05
```

```
PL/SQL execution elapsed time                    2.85          0.03
sequence load elapsed time                       0.45          0.00
hard parse (bind mismatch) elapsed time          0.03          0.00
hard parse (sharing criteria) elapsed time       0.03          0.00
inbound PL/SQL rpc elapsed time                  0.00          0.00
Java execution elapsed time                      0.00          0.00
failed parse (out of shared memory) elapsed t    0.00          0.00
connection management call elapsed time          0.00          0.00
PL/SQL compilation elapsed time                  0.00          0.00
```

不难看出，解析的时间占比是 29.9%，而硬解析所占比例很小，只有 0.25%。这种情况下，我认为优化软解析会取得比较好的效果。另外，由于每秒存在 2110 多个物理读操作，因此减少物理读操作也会对整体性能优化有较大的帮助。鉴于以上原因，我提出了如下优化建议：

- 将 SESSION_CACHED_CURSORS 参数设置为 200；
- 将 OPEN_CURSORS 参数加大为 3000；
- 如果物理内存充足，可将 DB Cache 加大至 5 ~ 10 GB。

他们根据上述建议调整了整个测试环境，修改了相应参数。第二天，我接到现场负责人的电话，进行调整后每秒交易数达到了 400 左右，但系统负载仍然只有 30%左右。不过这与客户每秒 600 个交易的要求仍差之甚远。他们希望我能够到现场进一步对系统进行调整。

我马上预订了机票，第二天下午就赶到了现场。在查看了测试期间采集的 AWR 报告后，我发现加大 DB Cache，设置 SESSION_CACHED_CURSORS 参数后，库缓存方面的争用大幅减少了，只在 TOP 5 EVENTS 的最后出现了一个库缓存的等待事件，占总等待的 2.3%。不过一些新的问题也随之出现了：

- LOG FILE SYNC 等待排到了第二位，占总等待的 19%；
- REDO LOG 日志每秒的生成量达到了 2 MB 多，按照这样计算，每分钟产生的 REDO 文件将超过 120 MB，而目前每个 REDO LOG 文件的大小仅为 100 MB，也就是说，不到 1 分钟就会产生 1 次日志切换；
- 在几张表上出现了严重的 ITL 争用；
- 系统使用了较为陈旧的 XP128 作为存储，其性能较差，而且所有的盘都是 RAID 5 的，写性能非常差。

经过上述分析，我决定采取如代码清单 18-1 所示的几项措施。

代码清单 18-1

```
----设置参数，解决几个 SQL 执行计划错误的问题
alter session set "_always_anti_join" =hash;

----设置一个 4 GB 的 KEEP POOL。
ALTER SYSTEM SET DB_CACHE_KEEP_SIZE=4g ;

----加大 LOG BUFFER 为 60 M
ALTER SYSTEM SET LOG_BUFFER=62914560 ;

----调整 REDO LOG 文件，将 REDO LOG 文件加大为 2GB
```

```
ALTER DATABASE DROP LOGFILE GROUP 1;
ALTER  DATABASE ADD LOGFILE THREAD 1
                 GROUP 1 ('/dev/vg_data04/rlv_data181') SIZE 2147483648;
 ----将日志切到 GROUP 1
ALTER DATABASE DROP LOGFILE GROUP 2;
ALTER DATABASE DROP LOGFILE GROUP 3;
ALTER  DATABASE ADD LOGFILE THREAD 1
                 GROUP 2 ('/dev/vg_data04/rlv_data182') SIZE 2147483648,
                 GROUP 3 ('/dev/vg_data04/rlv_data183') SIZE 2147483648;
ALTER  DATABASE ADD LOGFILE THREAD 1
                 GROUP 4 ('/dev/vg_data04/rlv_data184') SIZE 2147483648,
                 GROUP 5 ('/dev/vg_data04/rlv_data185') SIZE 2147483648,
                 GROUP 6 ('/dev/vg_data04/rlv_data186') SIZE 2147483648,
                 GROUP 7 ('/dev/vg_data04/rlv_data187') SIZE 2147483648;

----生成脚本，将 P*的表所相关的索引全部放入 KEEP POOL
select 'alter index '||index_name||' storage(buffer_pool keep);' from user_indexes
where table_name like 'P%'

----生成脚本，将 P*的表放入 KEEP POOL
 select 'alter table '||table_name||' storage(buffer_pool keep);' from user_tables
where table_name like 'P%'

----生成脚本，将 P*的表全部扫描一遍，使之在测试前全部放入 KEEP 池
 select 'select /*+ full(a) */ * from '||table_name ||' a;' from user_tables where
table_name like 'P%';

--加大几张表的 ITL

alter table aknmx pctfree 0 initrans 60;
alter table aknto pctfree 0 initrans 60;
alter table bjyrz pctfree 0 initrans 60;
alter table bdpal pctfree 0 initrans 60;
alter table adkmx pctfree 0 initrans 60;
```

调整后，我们马上进行了测试，发现性能有了明显的提升，平均每秒交易数提高到 920 左右。这时，在 AWR 报告中，LOG FILE SYNC 等待排到了第一位，虽然每次等待只有 4 毫秒，但无法继续增大压力测试，而此时 CPU 的使用率只有不到 80%。前几天，另外一个厂家的系统也在此环境下进行过测试，每秒交易数达到了 1100，但是 CPU 使用率只有 20%。客户对此表示怀疑，尽管交易数很高，而且事后审核也并未发现问题，但在这么大的交易量下，系统资源消耗却如此之小，这确实不大可信。

虽然我们都认为竞争对手存在作弊行为，但是并没有确凿的证据。但如果我们的测试数据能够超过对手，并且从测试结果中能够看到系统资源已经完全耗尽，那么既可以证明系统的吞吐能力强于对手，又可以证明对手存在舞弊行为。

从目前的情况来看，要想达到此目标，需要进一步加大 DB Cache，因此必须减少物理读操作的数量，但这还不足以使整个测试性能有质的飞跃，解决 LOG FILE SYNC 等待的问题才是关键。但从目前存储的性能来看，要想改善 REDO LOG 的写性能几乎是不可能的，LOG BUFFER 已经加大到了 60 MB，再继续增加也不会有太大的帮助。如何才能缩短 LOG FILE SYNC 等待的

时间呢?

突然，我想起了异步提交，这种提交模式无需等待 LGWR 进程将 LOG BUFFER 中的数据写入 REDO LOG 文件，可以直接返回提交成功。如果在生产系统中采用这种模式，一旦实例宕掉，可能导致部分已经提交的事务丢失。但在测试环境中，并不存在这个问题，实例宕掉的情况极少出现。即便出现了这种情况，也可以重新进行测试。于是我们又进行了一些微调，将系统的默认提交模式设置为异步非等待模式。在最终的测试结果中，每秒交易数达到了 1220，CPU 使用率也达到了 95%以上。这远远超出了竞争厂商作弊后的测试结果，不过我们在测试过程中也有些很小的“作弊”行为。

这个案例提到的处理方法，实际上是优化单个事务规模较小、提交很频繁的大并发系统时最为常用的方法。除了最后的异步提交技术，其他的优化手段都是十分常见的，大家可以在优化类似系统时参考使用。另外，异步提交技术并不仅仅适用于测试环境，在实际生产环境中，如果存在数据库实例宕机后的数据处理方案，那么就可以放心地使用该技术来提升大并发提交的性能。

18.2　IMP 导入性能问题的分析

公司的小杨打电话向我求助，一张表的 IMP 导入工作从 10 月 7 日就开始了，根据测算应该在 10 月 8 日 23 点左右完成，但直到现在（10 月 9 日上午 9 点）仍未完成。已经检查过 DBA_EXTENTS 视图，到目前为止该表一共导入了 18 GB 的数据，表的大小为 25 GB，共包含 2.5 亿多条记录。

很多 DBA 碰到这类问题都感到无从入手，即使分析出了问题原因，也不知道该如何处理。是重新导入，还是继续等待呢？到底还要多长时间才能完成导入？其实，这类问题的分析方法并不复杂，涉及的都是我们最为熟知的工具和知识。

首先，我问小杨是如何测算导入时间的，他告诉我是根据原表的扩展大小，然后通过 DBA_EXTENT 视图查看 1 小时大概的增长数量，估算出来的。原本预计导入操作 10 多个小时就可以完成，但实际上到现在已经过去 36 个小时了，仍有近 30%的数据没有完成导入。

分析这个问题时，首先需要明确 IMP 进程在等待什么，通过 V$SESSION_WAIT 视图可以看到该进程总是在等待 db file sequential read 事件。从这个等待事件的 p1、p2 参数可以看出，等待主要集中在索引上。于是我就问小杨，导入时是否提前建好了表和索引。

回答是肯定的，至此，这个问题就基本被定位了。对于一张大表来说，附带大量索引的导入操作速度会比普通导入操作下降数倍，而且随着索引叶节点分裂，在后期会变得更慢。其实，小杨也早就想到了这方面的问题，但在另外一台机器上进行测试时，这张表能够在 24 小时内完成导入，对于这次数据迁移，24 小时完成是可以接受的。为什么在这两台机器上性能的差别会如此之大呢?

这并不难理解，通过 AWR 报告的比对，我们很快就会发现，这套系统的 I/O 性能要低于测试系统。测试系统的 db file sequential read 等待事件响应时间在 4 毫秒左右，而当前系统的平均

响应时间为 9 毫秒左右。另外，当前系统的 LOG FILE SYNC 等待也要严重很多。I/O 性能的好坏对数据导入操作的性能影响是很大的。由于两套系统的 I/O 性能存在差异，出现这样的问题也就不足为奇了。

既然已经了解了 IMP 导入性能出现问题的原因，接下来就需要估算导入操作需要多长时间才可以完成。如果时间过长，就应该杀掉现在的导入进程，然后删除索引，重新进行导入操作。

由于之前进行过类似的导入操作，因此我们知道该表全部导入后，整个段大小约为 25 GB。而目前经过 36 个小时的导入，段大小为 18 GB，以此为依据，可以估算出导入所需的总时间为 $36 \div 18 \times 25=50$ 小时，也就是说，导入工作大概还需要 14 个小时才能完成。实际上，这种估算并不一定准确，因为导入操作的性能是递减的，而且这套系统使用的存储是共享的，半夜没有业务的时候和白天业务量较大的时候，存储负载是不同的。因此实际速度可能会更慢一些。这一点可以通过以下方法来验证，首先，生成一个 AWR 报告，查看这条 SQL 的 SQL_ID。

```
  Elapsed      CPU                  Elap per  % Total
   Time (s)   Time (s)  Executions   Exec (s)  DB Time    SQL Id
---------- ---------- ------------ ---------- ------- -------------
     3,509        312           77       45.6    89.7 7jwfg3d18v052
Module: imp@sjzzw31 (TNS V1-V3)
INSERT /*+NESTED_TABLE_SET_REFS+*/ INTO "XXXX_ITEM_OWE" ("XXXX_ITEM_ID", "ITEM_S
OURCE_ID", "SSSS_ID", "XXXX_ITEM_TYPE_ID", "CYCLE_ID", "XXXX_ID", "MMMM_DATE",
"INV_OFFER", "OFFER_ID", "OR_ITEM_ID") VALUES (:1, :2, :3, :4, :5,
```

通过 SQL_ID 生成 awrsqrpt 报告，用于进一步分析这条 SQL 的执行情况。

```
Stat Name                                Statement   Per Execution % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                         3,508,949       45,570.8    89.7
CPU Time (ms)                               312,280        4,055.6    51.7
Executions                                       77            N/A     N/A
Buffer Gets                              ##########      229,258.8    21.5
Disk Reads                                  417,661        5,424.2    90.7
Parse Calls                                       0            0.0     0.0
Rows                                      2,523,059       32,767.0     N/A
User I/O Wait Time (ms)                   3,198,675            N/A     N/A
Cluster Wait Time (ms)                       41,548            N/A     N/A
Application Wait Time (ms)                        0            N/A     N/A
Concurrency Wait Time (ms)                       67            N/A     N/A
Invalidations                                     0            N/A     N/A
Version Count                                     2            N/A     N/A
Sharable Mem(KB)                                 59            N/A     N/A
          -------------------------------------------------------------
```

可以看出，平均每次执行的行数（也就是每次插入的数据量）为 32767，这是因为 IMP 进程采用 BULK INSERT 方式插入数据，根据 BUFFER 的大小不同，每次插入的数据量也不同。我们可以在 Statement 列中看到，在 AWR 报告期间（1 小时）一共处理了 2523059 行。1 小时插入 250 万行数据，插入性能很差。通过同样的方法，生成 IMP 进程刚刚开始时的 awrsqrpt 报告：

```
Stat Name                                Statement   Per Execution % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                         1,980,963        5,027.8     5.3
CPU Time (ms)                             1,200,232        3,046.3     6.8
Executions                                      394            N/A     N/A
Buffer Gets                              ##########      201,683.2     6.9
Disk Reads                                      903            2.3     0.2
Parse Calls                                       1            0.0     0.0
Rows                                     ##########       32,767.0     N/A
User I/O Wait Time (ms)                      20,220            N/A     N/A
Cluster Wait Time (ms)                       20,987            N/A     N/A
Application Wait Time (ms)                        2            N/A     N/A
Concurrency Wait Time (ms)                   23,900            N/A     N/A
Invalidations                                     0            N/A     N/A
Version Count                                     1            N/A     N/A
Sharable Mem(KB)                                 30            N/A     N/A
          -------------------------------------------------------------
```

此报告中的 Rows 超长了，所以显示的都是#，不过我们可以通过 Per Execution 的数据来计算：执行次数与每次处理的记录数的乘积，就是总记录数。通过计算发现，刚开始插入时每小时插入的数据量为 $32767 \times 394 = 12910198$，这表明此时的插入速度可以达到每小时 1291 万条记录。较此数据，目前的插入性能已经下降了数倍。

从目前的情况来看，估计再过 10 个小时也无法完成插入操作，我们可以通过更为精确地计算来验证此观点。这张表的总记录数为 2.5 亿，目前已经完成了 18/25，还有 7/25 的数据未完成，即约 7000 万条记录未导入。如果按照目前的速度，每小时导入 250 万条记录，那么全部导入还需要 28 个小时。

如果我们删除所有的索引，然后重新导入，可能在 10 个小时内就能完成，因此必须终止导入操作，重新开始。

于是我们终止了当前操作，删除了表上的所有索引，然后截取（truncate）了相关表，重新进行导入操作，并且在导入时添加了 INDEXES = N 条件，避免索引生成。接下来，手工编写索引创建脚本，并行创建索引。5 个小时后，导入工作顺利完成。

这个案例并不复杂，是很多 DBA 都经常遇到的。通过此案例，老白介绍了一种利用 AWR 报告计算 IMP 速度的方法，该方法已经多次为老白制定决策提供了帮助。

18.3 并行操作为什么无法执行

以前，在一次系统割接时，我们曾碰到过一个十分奇怪的现象。由于要进行系统迁移，很多大表在数据导入时都没有创建索引，因此导入结束后需要重建索引。为了加快索引的创建速度，需要进行并行创建。虽然我们在创建索引的脚本中加入了 PARALLEL 40 子句，但实际上，创建索引的操作仍是串行的。

这是一套 64 核的系统，并行创建索引可以成倍地提高速度，而无法使用并行严重影响了割接前的准备工作，因此必须尽快查清问题原因。首先，我们需要检查并行的相关参数设置。

```
SQL> show parameter parallel

NAME                                 TYPE        VALUE
------------------------------------ ----------- --------------------------
fast_start_parallel_rollback         string      LOW
parallel_adaptive_multi_user         boolean     TRUE
parallel_automatic_tuning            boolean     FALSE
parallel_execution_message_size      integer     2152
parallel_Instance_group              string      XXXX31
parallel_max_servers                 integer     1000
parallel_min_percent                 integer     0
parallel_min_servers                 integer     10
parallel_server                      boolean     TRUE
parallel_server_Instances            integer     2
parallel_threads_per_cpu             integer     2
recovery_parallelism                 integer     0
```

可以看出，PARALLEL 相关的参数设置并无问题，PARALLEL_MAX_SERVERS 参数为 1000，PARALLEL_MIN_SERVERS 参数为 10。通过 ps 命令可以看出，目前系统只启动了 10 个并行进程，也就是 PARALLEL_MIN_SERVERS 参数指定的数量。

```
oracle@test31:/oracle$ ps -ef|grep p0
  oracle 13044     1  0  Oct 20  ?         0:04 ora_p008_test31
  oracle 13038     1  0  Oct 20  ?         0:04 ora_p005_test31
  oracle 13029     1  0  Oct 20  ?         0:04 ora_p003_test31
  oracle 13027     1  0  Oct 20  ?         0:04 ora_p002_test31
  oracle  6425     1  0  Oct 18  ?         0:08 ora_psp0_test31
  oracle 13031     1  0  Oct 20  ?         0:04 ora_p004_test31
  oracle 13025     1  0  Oct 20  ?         0:04 ora_p001_test31
  oracle 13040     1  0  Oct 20  ?         0:04 ora_p006_test31
  oracle 13023     1  0  Oct 20  ?         0:04 ora_p000_test31
  oracle 13046     1  0  Oct 20  ?         0:04 ora_p009_test31
  oracle 13042     1  0  Oct 20  ?         0:04 ora_p007_test31
```

从 ps 的结果来看，并行进程的启动是正常的。在 ALERT LOG 日志中也没有出现相关的错误或者警告信息。这个问题确实有点奇怪，为了尽快定位问题，我们创建了如下所示的测试环境。

```
create table xuji_test tablespace sysaux as select * from dba_objects ;
alter table xuji_test parallel 20;
select count(*) from xuji_test;
```

接下来，通过 DBA_OBJECTS 视图创建一张包含 6 万多条记录的表 xuji_test，然后将这张表的并行度设置为 20，并进行一次 COUNT(*)操作，SQL 执行后，从 V$SQLAREA 视图中找到这条 SQL 的 SQL_ID，其执行计划如下：

```
SQL> select * from
table(dbms_xplan.display_cursor('8sj2h9nsq7s4h',null,'ADVANCED'));
select count(*) from xuji_test
Plan hash value: 3609358487
-------------------------------------------------------------------------------
| Id  | Operation              | Name       | Rows  |  TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |            |       |      |      |            |
```

```
|   1 | SORT AGGREGATE          |           |     1 |       |       |            |
|   2 | PX COORDINATOR          |           |       |       |       |            |
|   3 | PX SEND QC (RANDOM)     | :TQ10000  |     1 |Q1,00 | P->S | QC (RAND)  |
|   4 | SORT AGGREGATE          |           |     1 |Q1,00 | PCWP |            |
|   5 | PX BLOCK ITERATOR       |           | 61059 |Q1,00 | PCWC |            |
|*  6 | TABLE ACCESS FULL       | XUJI_TEST | 61059 |Q1,00 | PCWP |            |
-------------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------
   6 - SEL$1 / XUJI_TEST@SEL$1
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('10.2.0.4')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------
      FULL(@"SEL$1" "XUJI_TEST"@"SEL$1")
      END_OUTLINE_DATA
  */
Predicate Information (identified by operation id):
---------------------------------------------------
   6 - access(:Z>=:Z AND :Z<=:Z)

Column Projection Information (identified by operation id):
-----------------------------------------------------------
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------
   1 - (#keys=0) COUNT()[22]
   2 - SYS_OP_MSR()[10]
   3 - (#keys=0) SYS_OP_MSR()[10]
   4 - (#keys=0) SYS_OP_MSR()[10]
```

从执行计划来看，好像这条 SQL 是并行查询的。不过从 ps 结果来看，并行进程并无改变：

```
oracle@test31:/oracle$ ps -ef|grep p0
  oracle 13044     1  0  Oct 20  ?         0:04 ora_p008_test31
  oracle 13038     1  0  Oct 20  ?         0:04 ora_p005_test31
  oracle 13029     1  0  Oct 20  ?         0:04 ora_p003_test31
  oracle 13027     1  0  Oct 20  ?         0:04 ora_p002_test31
  oracle  6425     1  0  Oct 18  ?         0:08 ora_psp0_test31
  oracle 13031     1  0  Oct 20  ?         0:04 ora_p004_test31
  oracle 13025     1  0  Oct 20  ?         0:04 ora_p001_test31
  oracle 13040     1  0  Oct 20  ?         0:04 ora_p006_test31
  oracle 13023     1  0  Oct 20  ?         0:04 ora_p000_test31
  oracle 13046     1  0  Oct 20  ?         0:04 ora_p009_test31
  oracle 13042     1  0  Oct 20  ?         0:04 ora_p007_test31
```

为了进一步确认并行查询是否发生，我们对一张记录数为 3 亿的大表进行了查询。

```
select /*+ full(a) parallel( a 50) */ from xxxx a;
```

此查询执行了 10 分钟，在 SQL 执行期间查询视图 V$PX_SESSION：

```
SQL> select * from v$px_session;

未选定行
```

从查询结果来看，无并行进程存在，因此可以确定并行查询并未发生。下一步该如何处理呢？看来只能进行跟踪了，跟踪并行查询可以使用隐含参数_PX_TRACE。关于如何使用该参数来分析并行查询，可以参考 Metalink 文档 How to Use _PX_TRACE to Check Whether Parallelism is Used [ID 400886.1]。

首先在会话中设置_PX_TRACE 参数：

```
SQL> alter session set "_px_trace"="compilation","execution","messaging"
 2 /
会话已更改。
```

上述设置的含义是在 SQL 编译、执行和并行执行消息传递活动发生时进行跟踪。设置好参数后，执行如下查询操作：

```
SQL> select count(*) from xuji_test;
COUNT(*)
 61059
```

SQL 执行结束后，在 udump 目录下找到这个 trace 文件，其内容如下：

```
*** ACTION NAME:() 2011-10-20 20:43:29.336
*** MODULE NAME:(sqlplus@test31 (TNS V1-V3)) 2011-10-20 20:43:29.336
*** SERVICE NAME:(SYS$USERS) 2011-10-20 20:43:29.336
*** SESSION ID:(2720.703) 2011-10-20 20:43:29.336
kkfdapdml
 pgadep:0 pdml mode:0 PQ allowed DML allowed not autonomous => not allowed
kxfplist
 Getting Instance info for open group
kxfralo
 serial - Instance group has no open members
~
~
```

这段信息中的第一句 pgadep:0 是每个 trace 文件都包含的，不必留意。下面的 kxfplist 和 kxfralo 这两句十分重要，其含义为查找实例的并行组（parallel group），判断本会话是否属于开放的并行组。如果会话的 PARALLEL_INSTANCE_GROUP 参数指定的并行组在某个实例中未设置，那么就不能使用并行查询。从 kxfralo 的结果可以看出，最终选择的执行方式是串行（serial），而选择串行的原因是“Instance group has no open members”，也就是说当前 PARALLEL_INSTANCE_GROUP 参数指定的组不属于 INSTANCE_GROUPS 参数指定的组。难道是 PARALLEL_INSTANCE_GROUP 参数设置的问题吗？我们再来看看下面的信息：

```
SQL> show parameter Instance_group
```

```
NAME TYPE VALUE
------------------------------------ ----------- ------------------------------
Instance_groups string test3,test31
parallel_Instance_group string test31
```

这里好像并没有什么问题，PARALLEL_INSTANCE_GROUP 设置为 test31，而实例的组设置为 test3 和 test31。尽管根据跟踪结果定位的并行查询无法执行的原因是 PARALLEL_INSTANCE_GROUP 参数设置存在问题，但从参数本身来看，并没有任何问题，难道碰到灵异事件了？其实 Oracle 根本不可能存在灵异事件，肯定是我们忽略了什么。通过一种简单有效的方法可以验证参数设置是否存在问题，即查看正常数据库中的参数设置。

```
SQL> show parameter Instance_group
NAME TYPE VALUE
------------------------------------ ----------- ------------------------------
Instance_groups string test1, test11
parallel_Instance_group string test11
```

从表面来看，好像并没有不同。但经过 5 分钟的反复比对，我终于发现正确的 INSTANCE_GROUPS 参数的两个组之间有一个空格，这可能就是问题所在。下一步，我们来验证这个空格是否和参数设置不同有关。在这两个系统上，分别生成一个 pfile 文件，来查看参数。

```
create pfile='/tmp/init.ora' from spfile
```

问题系统的参数设置为：

```
Instance_groups='test3,test31'
```

正常系统的参数设置为：

```
Instance_groups='test1','test11'
```

在参数文件中，结果就更清晰了。一个是单引号括住了两个组；另一个是每个组分别用单引号括起来，并用逗号分割。第一种配置实际上是将 INSTANCE_GROUPS 参数设置为一个名为 test3,test31 的组（逗号是组成组名的合法字符），可以通过以下示例来验证。

```
SQL> alter session set "_px_trace"="compilation","execution","messaging";
会话已更改。
SQL> alter session set parallel_Instance_group='test3,test31';
会话已更改。
SQL> select count(*) from xuji_test;
COUNT(*)
----------
 61059
```

于是，我们将会话的 PARALLEL_INSTANCE_GROUP 参数设置为 test3,test31，使之符合并行查询的条件。令人兴奋的是，trace 文件发生了改变。

```
*** ACTION NAME:() 2011-10-20 20:53:48.616
*** MODULE NAME:(sqlplus@test31 (TNS V1-V3)) 2011-10-20 20:53:48.616
*** SERVICE NAME:(SYS$USERS) 2011-10-20 20:53:48.616
*** SESSION ID:(4121.314) 2011-10-20 20:53:48.616
kkfdapdml
 pgadep:0 pdml mode:0 PQ allowed DML allowed not autonomous => not allowe d
```

```
kxfplist
 Getting Instance info for open group
kxfrSysInfo
 DOP trace -- compute default DOP from system info
 # Instance alive = 1 (kxfrsnins)
kxfrDefaultDOP
 DOP Trace -- compute default DOP
 # CPU = 64
 Threads/CPU = 2 ("parallel_threads_per_cpu")
 default DOP = 128 (# CPU * Threads/CPU)
 default DOP = 128 (DOP * # Instance)
kxfrSysInfo
 system default DOP = 128 (from kxfrDefaultDOP())
kxfralo
 DOP trace -- requested thread from best ref obj = 20 (from kxfrIsBestRef
 ())
kxfralo
 threads requested = 20 (from kxfrComputeThread())
kxfralo
 adjusted no. threads = 20 (from kxfrAdjustDOP())
kxfralo
 about to allocate 20 slaves
kxfrAllocSlaves
 DOP trace -- call kxfpgsg to get 20 slaves
kxfpgsg
 num server requested = 20
kxfplist
 Getting Instance info for open group
kxfpiinfo
 inst[cpus:mxslv]
 1[64:1000]
kxfpclinfo
 inst(load:user:pct:fact)aff
 1(3:0:100:2133)
kxfpAdaptDOP
 Requested=20 Granted=20 Target=512 Load=3 Default=128 users=0 sets=1
kxfpgsg
getting 1 sets of 20 threads, client parallel query execution flg=0x30
 Height=20, Affinity List Size=0, inst_total=1, coord=1
 Insts 1
 Threads 20
kxfpg1srv
 trying to get slave P000 on Instance 1
kxfpg1sg
 Got It. 1 so far.
kxfpg1srv
 trying to get slave P001 on Instance 1
kxfpg1sg
 Got It. 2 so far.
kxfpg1srv
 trying to get slave P002 on Instance 1
kxfpg1sg
```

```
 Got It. 3 so far.
kxfpg1srv
 trying to get slave P003 on Instance 1
kxfpg1sg
 Got It. 4 so far.
kxfpg1srv
 trying to get slave P004 on Instance 1
kxfpg1sg
 Got It. 5 so far.
kxfpg1srv
 trying to get slave P005 on Instance 1
kxfpg1sg
 Got It. 6 so far.
kxfpg1srv
 trying to get slave P006 on Instance 1
kxfpg1sg
 Got It. 7 so far.
kxfpg1srv
 trying to get slave P007 on Instance 1
kxfpg1sg
 Got It. 8 so far.
kxfpg1srv
 trying to get slave P008 on Instance 1
kxfpg1sg
 Got It. 9 so far.
kxfpg1srv
 trying to get slave P009 on Instance 1
...
```

看来我们的猜测是正确的，问题解决了。由于修改 INSTANCE_GROUPS 参数需要重启实例，因此我们可以通过会话级修改 PARALLEL_INSTANCE_GROUP 参数来规避这个问题，等到可以重启实例时再彻底解决该问题。

后来我在 Metalink 上找到了一篇相关的文档（After changing the init parameter Instance _GROUPS, queries are no longer being executed in parallel [ID 750645.1]），正好是讲述这个问题的。这篇文档指出，INSTANCE_GROUPS 参数设置错误将会导致并行执行无法正常工作。

```
Changed the initialization parameter settings for the parameters Instance_GROUPS and
PARALLEL_Instance_GROUP.  Now the parameters are as follows:
*.Instance_groups='MYRAC,MYRAC1,MYRAC2,MYRAC3'
MYRAC1.parallel_Instance_group='MYRAC1'
MYRAC2.parallel_Instance_group='MYRAC2'
MYRAC3.parallel_Instance_group='MYRAC3'

After restarting the Instances, parallel execution is disabled on all Instances.
Parallel query processes do not get spawned even when the execution plan shows parallel.
```

此外，文章还指出，要解决这一问题，需要对 INSTANCE_GROUPS 参数进行如下调整：

```
1. change the value of Instance_groups in the pfile or spfile
eg for spfile:
alter system set Instance_groups='MYRAC','MYRAC1','MYRAC2','MYRAC3' SCOPE=SPFILE
SID='*' ;
```

```
2. restart each Instance one at a time (to avoid downtime)

You should now be able to execute queries in parallel again.
```

这个案例看起来十分简单，并没有什么技术含量。不过如果第一次碰到这样的案例，可能就会觉得这是一个灵异事件。因此，首先我们要明确，任何不正常的事件肯定存在其错误的地方，只是有些错误十分隐秘，不太容易被察觉而已。碰到这样的问题时，要采取主动的手段去进一步分析，进行跟踪是最佳的分析方法，另外也还可以采用排除法帮助分析。

第 19 章

SQL 优化

19

SQL 优化一直是大多数 DBA 十分头痛的问题，很多 DBA 一直不敢涉猎这个领域。这是因为绝大多数 DBA 都没有应用开发的经验，另外，部分 DBA 甚至没有书写复杂 SQL 的经验。

实际上，SQL 优化并不是一件十分复杂的事情，而只是一项比较繁琐的工作。只要我们掌握了 SQL 分析的方法，并且有足够的耐心，就能够成为一个 SQL 优化的高手。本章将通过几个案例介绍 SQL 分析和优化的方法，大家可以通过这些案例学习到相关的分析技巧，并进行一些 SQL 优化实践。在完成了数十条甚至上百条 SQL 的分析优化工作后，你很可能就会成为一名 SQL 优化高手。

19.1　一个常用的 SQL 优化方法

如果我们在 AWR 报告中发现某条 SQL 的开销很大，该如何处理呢？比如，看到如下信息：

```
SQL ordered by Gets                   DB/Inst: SJZZW3/sjzzw31  Snaps: 560-561
-> Resources reported for PL/SQL code includes the resources used by all SQL
   statements called by the code.
-> Total Buffer Gets:   1,234,841,780
-> Captured SQL account for      83.2% of Total

                              Gets              CPU        Elapsed
  Buffer Gets   Executions    per Exec   %Total   Time (s)   Time (s)    SQL Id
-------------- ------------ ------------ -------- --------- --------- ------------
   372,748,885   25,348       14,705.3    30.2    3474.04    3582.40 bxta7sv1fzd4f
Module: send_direct@wxjfapp1 (TNS V1-V3)
 SELECT S_SEND_ID, nvl(S_ID, 0) S_ID, nvl(A_NBR, '0')
A_NBR, nvl(BAL, 0) BAL, nvl(AM, 0) AMO,
 nvl(E_DATE, '20001001') E_DATE, M_T_ID, STATE,
 nvl(M_CONTENT, ' ') M_CONTENT, nvl(B_MODE_ID, 2) B_MODE
```

这条 SQL 在一个小时内执行了 25 348 次，占整个系统的 BUFFER GET 总量的 30.2%。它对系统资源的消耗是非常大的，如果能够优化，将使系统总体的性能获得大幅提升。对于这样一条典型的 SQL，我们该怎样进行优化呢？首先需要格式化 SQL 语句，使之更为清晰。格式化 SQL 可以借助工具来完成，PL/SQL DEVELOPER、TOAD 以及 Oracle SQL DEVELOPER 都有格式化 SQL 的工具。如果找不到合适的工具，也可以手工进行格式化。手工格式化是老白常用的方法。

在格式化 SQL 的时候，SELECT 语句后面的字段一般不是很关键，最关键的是 FROM 语句后面的表以及 WHERE 语句后面的条件。

在文本格式的 AWR 报告中，我们看到的 SQL 往往是不完整的（在 HTML 格式的文件中能够看到较为完整的 SQL），不过可以通过 awrsqrpt 工具来生成更为详细的信息。在 awrsqrpt 报告中找到完整的 SQL 语句，然后对其进行格式化：

```
SELECT ...
  FROM
     (SELECT A.*, B.C_SERVICE
       FROM AAAAAAAA A, BBBBBBBB B
       WHERE
        A.STATE = :1 AND A.NETWORK_ID in
          (select to_number(attr_value)
           from a_sms_attr where attr_type = 1) AND
        A.MSG_TYPE_ID not in (2, 22, 23, 24, 25, 27) AND
        A.SERV_ID >= :2 AND
        A.SERV_ID <= :3 AND
        A.MSG_TYPE_ID = B.MSG_TYPE_ID AND
        B.STATE = '10A' AND
        f_get_valid_send_sms_time(B.valid_time) = 0 AND
        SYSDATE >= NVL(A.SEND_SUCCESS_DATE, SYSDATE - B.SM_INTERVAL / 24) +
B.SM_INTERVAL / 24
    ORDER BY NVL(A.GENERATE_FLAG, 0), B.S_PRIORITY, B.PRIORITY, A.S_SEND_ID,
             A.SERV_ID, NVL(A.SEND_SUCCESS_DATE, TO_DATE('19000101', 'YYYYMMDD
             ')))
            WHERE ROWNUM <= :4
```

可以看出，A 表上的过滤条件比较多，其中，A.STATE=:1 以及 A.SERV_ID>=:2 AND A.SERV_ID<=:3 都使用了绑定变量。如果需要了解该 SQL 中绑定变量的取值情况，可以进行如下操作：

```
SQL> select * from
table(dbms_xplan.display_cursor('bxta7sv1fzd4f',null,'ADVANCED'));
...此处省略 SQL 详细文本...
Plan hash value: 3446859354
-----------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows | Bytes |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |      |       |
|*  1 | COUNT STOPKEY                       |                   |      |       |
|*  2 | VIEW                                |                   |  1   |  751  |
|*  3 | FILTER                              |                   |      |       |
|   4 | SORT ORDER BY                       |                   |  1   |  255  |
|*  5 | HASH JOIN SEMI                      |                   |  1   |  255  |
|   6 | NESTED LOOPS                        |                   |  1   |  250  |
|*  7 | TABLE ACCESS BY GLOBAL INDEX ROWID  | S_SEND            |  1   |  210  |
|*  8 | INDEX RANGE SCAN                    | IDX_SMS_SEND_SERV|  2   |       |
|*  9 | TABLE ACCESS BY INDEX ROWID         | S_SEND_TYPE       |  1   |   40  |
|* 10 | INDEX UNIQUE SCAN                   | PK_S_SEND_TYPE_ID|  1   |       |
|* 11 | TABLE ACCESS FULL                   | A_SMS_ATTR        |  3   |   15  |
-----------------------------------------------------------------------------------
```

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$7EB7D0CF
   2 - SEL$2A90E61B / from$_subquery$_001@SEL$1
   3 - SEL$2A90E61B
   7 - SEL$2A90E61B / A@SEL$2
   8 - SEL$2A90E61B / A@SEL$2
   9 - SEL$2A90E61B / B@SEL$2
  10 - SEL$2A90E61B / B@SEL$2
  11 - SEL$2A90E61B / A_SMS_ATTR@SEL$3

Outline Data

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('10.2.0.4')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$2A90E61B")
      PULL_PRED(@"SEL$1" "from$_subquery$_001" 1)
      OUTLINE_LEAF(@"SEL$7EB7D0CF")
      OUTLINE(@"SEL$BE5C8E5F")
      UNNEST(@"SEL$3")
      OUTLINE(@"SEL$1")
      OUTLINE(@"SEL$2")
      OUTLINE(@"SEL$3")
      NO_ACCESS(@"SEL$7EB7D0CF" "from$_subquery$_001"@"SEL$1")
      INDEX_RS_ASC(@"SEL$2A90E61B" "A"@"SEL$2" ("S_SEND"."SERV_ID"))
      INDEX_RS_ASC(@"SEL$2A90E61B" "B"@"SEL$2" ("S_SEND_TYPE"."MSG_TYPE_ID"))
      FULL(@"SEL$2A90E61B" "A_SMS_ATTR"@"SEL$3")
      LEADING(@"SEL$2A90E61B" "A"@"SEL$2" "B"@"SEL$2" "A_SMS_ATTR"@"SEL$3")
      USE_NL(@"SEL$2A90E61B" "B"@"SEL$2")
      USE_HASH(@"SEL$2A90E61B" "A_SMS_ATTR"@"SEL$3")
      END_OUTLINE_DATA
  */

Peeked Binds (identified by position):
--------------------------------------

   1 - :1 (NUMBER): 0
   2 - :2 (NUMBER): 101112894255
   3 - :3 (NUMBER): 101113864255
   4 - :4 (NUMBER): 1000
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=:4)
   2 - filter("F_GET_VALID_SEND_SMS_TIME"("VALID_TIME")=0)
   3 - filter(:2<=:3)
   5 - access("A"."NETWORK_ID"=TO_NUMBER("ATTR_VALUE"))
   7 - filter(("A"."STATE"=:1 AND "A"."MSG_TYPE_ID"<>2 AND "A"."MSG_TYPE_ID"<>22 AND
              "A"."MSG_TYPE_ID"<>23 AND "A"."MSG_TYPE_ID"<>24 AND "A"."MSG_TYPE_ID"
              <>25 AND "A"."MSG_TYPE_ID"<>27))
   8 - access("A"."SERV_ID">=:2 AND "A"."SERV_ID"<=:3)
```

```
   9 - filter(("B"."STATE"='10A' AND NVL("A"."SEND_SUCCESS_DATE",SYSDATE@!-"B"."SM_
              INTERVAL"/24)+"B". "SM_INTERVAL"/24<=SYS DATE@!))
  10 - access("A"."MSG_TYPE_ID"="B"."MSG_TYPE_ID")
       filter(("B"."MSG_TYPE_ID"<>2 AND "B"."MSG_TYPE_ID"<>22 AND "B"."MSG_TYPE_ID"
              <>23 AND "B"."MSG_TYPE_ID"<>24 AND "B"."MSG_TYPE_ID"<>25 AND "B"."MSG_
              TYPE_ID"<>27))
  11 - filter("ATTR_TYPE"=1)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 -"SMS_SEND_ID"[NUMBER,22], "SERV_ID"[NUMBER,22], "ACC_NBR"[VARCHAR2,32],"BALANCE"
      [NUMBER,22], "AMOUNT" [NUMBER,22], "EXP_DATE"[VARCHAR2,8], "MSG_TYPE_ID"[NUMBER,
      22], "STATE"[NUMBER,22], "MSG_CONTENT"[VARCHAR2,1024], "BILLING_MODE_ID"[NUMBER,
      22], "NETWORK_ID"[NUMBER,22], "ACCT_CREDIT"[NUMBER,22],"SOURCE_SERV_ID"[NUMBER,22],
      "CALL_SERVICE"[VARCHAR2,30]
   2 -"SMS_SEND_ID"[NUMBER,22], "SERV_ID"[NUMBER,22], "ACC_NBR"[VARCHAR2,32],"BALANCE"
      [NUMBER,22], "AMOUNT"[NUMBER,22],
      "EXP_DATE"[VARCHAR2,8], "MSG_TYPE_ID"[NUMBER,22], "STATE"[NUMBER,22],"MSG_CONTENT"
      [VARCHAR2,1024], "BILLING_MODE_ID"[NUMBER,22], "NETWORK_ID"[NUMBER,22], "ACCT_
      CREDIT" [NUMBER,22], "SOURCE_SERV_ID"[NUMBER,22], "CALL_SERVICE"[VARCHAR2,30],
      "VALID_TIME"[VARCHAR2,128]
   3 -"A"."SMS_SEND_ID"[NUMBER,22],"A"."SERV_ID"[NUMBER,22],"B"."VALID_TIME"[VARCHAR2,
      128], "B"."CALL_SERVICE"[VARCHAR2,30], "A"."ACC_NBR"[VARCHAR2,32], "A"."BALANCE"
      [NUMBER,22], "A"."AMOUNT"[NUMBER,22], "A"."EXP_DATE"[VARCHAR2,8],"A"."MSG_TYPE_ID"
      [NUMBER,22], "A"."STATE"[NUMBER,22], "A"."MSG_CONTENT"[VARCHAR2,1024], "A".
      "BILLING_MODE_ID"[NUMBER,22], "A"."NETWORK_ID"[NUMBER,22], "A"."ACCT_CREDIT"
      [NUMBER,22], "A"."SOURCE_SERV_ID"[NUMBER,22]
   4- (#keys=6) NVL("A"."GENERATE_FLAG",0)[22], "B"."SMS_PRIORITY"[NUMBER,22], "B".
      "PRIORITY"[NUMBER,22], "A"."SMS_SEND_ID"[NUMBER,22], "A"."SERV_ID"[NUMBER,22],
      NVL("A"."SEND_SUCCESS_DATE",TO_DATE(' 1900-01-01 00:00:00', 'syyyy-mm-dd hh24:
      mi:ss'))[7], "B"."VALID_TIME"[VARCHAR2,128], "B"."CALL_SERVICE"[VARCHAR2,30],
      "A"."ACC_NBR"[VARCHAR2,32], "A"."BALANCE"[NUMBER,22],"A"."AMOUNT"[NUMBER,22],
      "A"."EXP_DATE"[VARCHAR2,8], "A"."MSG_TYPE_ID"[NUMBER,22],"A"."STATE"[NUMBER,
      22], "A"."MSG_CONTENT"[VARCHAR2,1024], "A"."BILLING_MODE_ID"[NUMBER,22], "A".
      "NETWORK_ID"[NUMBER,22], "A"."ACCT_CREDIT"[NUMBER,22], "A"."SOURCE_SERV_ID"
      [NUMBER,22]
   5 -(#keys=1) "A"."NETWORK_ID"[NUMBER,22], "A"."SMS_SEND_ID"[NUMBER,22], "A".
      "SERV_ID"[NUMBER,22], "A"."ACC_NBR"[VARCHAR2,32], "A"."BALANCE"[NUMBER,22],
      "A"."AMOUNT"[NUMBER,22], "A"."EXP_DATE"[VARCHAR2,8], "A"."MSG_TYPE_ID" [NUMBER,
      22], "A"."STATE"[NUMBER,22], "A"."SEND_SUCCESS_DATE"[DATE,7], "A"."MSG_CONTENT"
      [VARCHAR2,1024], "A"."BILLING_MODE_ID"[NUMBER,22], "A"."ACCT_CREDIT"[NUMBER,22],
      "A"."GENERATE_FLAG"[NUMBER,22], "A"."SOURCE_SERV_ID"[NUMBER,22], "B"."PRIORITY"
      [NUMBER,22], "B"."SMS_PRIORITY"[NUMBER,22], "B"."CALL_SERVICE"[VARCHAR2,30],
      "B"."VALID_TIME"[VARCHAR2,128] 6 - "A"."SMS_SEND_ID"[NUMBER,22], "A"."SERV_ID"
      [NUMBER,22], "A"."ACC_NBR"[VARCHAR2,32], "A"."BALANCE"[NUMBER,22], "A"."AMOUNT"
      [NUMBER,22], "A"."EXP_DATE"[VARCHAR2,8], "A"."MSG_TYPE_ID"[NUMBER,22], "A".
      "STATE" [NUMBER,22], "A"."SEND_SUCCESS_DATE"[DATE,7], "A"."MSG_CONTENT"[VARCHAR2,
      1024], "A"."BILLING_MODE_ID"[NUMBER,22], "A"."NETWORK_ID"[NUMBER,22],"A"."ACCT_
      CREDIT"[NUMBER,22], "A"."GENERATE_FLAG"[NUMBER,22],"A"."SOURCE_SERV_ID"[NUMBER,
      22],"B"."PRIORITY"[NUMBER,22],"B"."SMS_PRIORITY"[NUMBER,22],"B"."CALL_SERVICE"
      [VARCHAR2,30], "B"."VALID_TIME"[VARCHAR2,128]
   7 -"A"."SMS_SEND_ID"[NUMBER,22], "A"."SERV_ID"[NUMBER,22],"A"."ACC_NBR"[VARCHAR2,32],
      "A"."BALANCE"[NUMBER,22], "A"."AMOUNT"[NUMBER,22], "A"."EXP_DATE"[VARCHAR2,8],
```

```
      "A"."MSG_TYPE_ID"[NUMBER,22], "A"."STATE"[NUMBER,22], "A"."SEND_SUCCESS_DATE"
      [DATE,7], "A"."MSG_CONTENT"[VARCHAR2,1024], "A"."BILLING_MODE_ID"[NUMBER,22],
      "A"."NETWORK_ID"[NUMBER,22], "A"."ACCT_CREDIT"[NUMBER,22], "A"."GENERATE_FLAG"
      [NUMBER,22], "A"."SOURCE_SERV_ID"[NUMBER,22]
   8- "A".ROWID[ROWID,10], "A"."SERV_ID"[NUMBER,22]
   9- "B"."PRIORITY"[NUMBER,22], "B"."SMS_PRIORITY"[NUMBER,22], "B"."CALL_SERVICE"
[VARCHAR2,30], "B"."VALID_TIME"[VARCHAR2,128]
  10 - "B".ROWID[ROWID,10]
  11 - "ATTR_VALUE"[VARCHAR2,12]
138 rows selected.
```

select * from table (dbms_xplan.display_cursor ('SQL Id',null,'ADVANCED'))语句可以返回 SQL 执行计划的详细信息和具体细节，其中也包含了绑定变量的信息。这条 SQL 中 4 个绑定变量的详细信息如下：

```
Peeked Binds (identified by position):
--------------------------------------

   1 - :1 (NUMBER): 0
   2 - :2 (NUMBER): 101112894255
   3 - :3 (NUMBER): 101113864255
   4 - :4 (NUMBER): 1000
```

其中，STATE 字段的取值是 0，SERV_ID 的上下边界是 10 112 894 255 ~ 10 113 864 255，而 ROWNUM 是小于 1000 的。从这几个数据可以看出，SERV_ID 的过滤条件范围较为广泛。目前的 SQL 执行计划中，对 A 表的访问是对 SERV_ID 的索引进行范围扫描，符合条件的记录查找 A 表再进行扫描，然后和 B 表做嵌套循环，这两个表连接的结果再和 C 表做散列连接。这个执行计划是否合理呢？我们来看看 awrsqrpt 报告中的 SQL 统计信息：

```
Plan Statistics                        DB/Inst: SJZZW3/sjzzw31  Snaps: 560-561
-> % Total DB Time is the Elapsed Time of the SQL statement divided
   into the Total Database Time multiplied by 100

Stat Name                                 Statement   Per Execution    % Snap
---------------------------------------- ------------ ---------------- --------
Elapsed Time (ms)                           3,582,402           141.3      8.7
CPU Time (ms)                               3,474,041           137.1     12.2
Executions                                     25,348             N/A      N/A
Buffer Gets                                ##########        14,705.3     30.2
Disk Reads                                         31             0.0      0.0
Parse Calls                                    25,347             1.0      0.1
Rows                                            1,445             0.1      N/A
User I/O Wait Time (ms)                           261             N/A      N/A
Cluster Wait Time (ms)                             14             N/A      N/A
Application Wait Time (ms)                          0             N/A      N/A
Concurrency Wait Time (ms)                        404             N/A      N/A
Invalidations                                       0             N/A      N/A
Version Count                                       1             N/A      N/A
Sharable Mem(KB)                                   68             N/A      N/A
          -------------------------------------------------------------------
```

在这里我们可以看到，平均每次执行的 BUFFER GET 为 14 705 次，这是很大的数字，而平均每次执行返回的记录数是 0.1。这说明大多数的 SQL 执行都返回了 0 条记录，也就说明可能存

在过滤性很强的条件。于是，我们首先想到了 STATE 字段，从字面含义来看，该字段是一个值域范围很小的字段，一般来说过滤性不会太好。不过是否存在一种可能，就是这个字段是倾斜的，正好执行这条 SQL 所对应的 STATE 值在 A 表中的记录数很少。针对此猜测，我们可以进行验证：

```
SQL> select count(*),state from aaaaaa group by state;

  COUNT(*)      STATE
---------- ----------
     21247          1
    100797          0
```

可以看出，这个字段只有两个取值，而且确实是倾斜的，不过刚才在绑定变量中我们已经看到了，STATE=0，正好是记录数较多的那个取值。看来 STATE 字段并不是一个很好的过滤条件，那么下一步就需要查看其他的过滤条件：

```
   WHERE
    A.STATE = :1 AND A.NETWORK_ID in
      (select to_number(attr_value)
       from cccccc where attr_type = 1) AND
    A.MSG_TYPE_ID not in (2, 22, 23, 24, 25, 27) AND
    A.SERV_ID >= :2 AND
    A.SERV_ID <= :3 AND
```

如果 STATE 再加上 MSG_TYPE_ID，这样的组合条件的效果如何呢？

```
SQL> SELECT count(*)
  2         FROM AAAAAA A
  3         WHERE
  4         A.STATE = 0 AND
            A.MSG_TYPE_ID not in (2, 22, 23, 24, 25, 27);
  COUNT(*)
----------
      1830
```

结果集还是有 1830 条记录，因此效果仍不够理想，这里面还有一个 A.NETWORK_ID 的过滤条件，不过此过滤条件是针对一个子查询的 IN 操作。幸运的是，这个子查询是一个固定条件的查询，我们来看看它的结果：

```
SQL> select to_number(attr_value)
  2             from OWNER1.a_sms_attr where attr_type = 1;

TO_NUMBER(ATTR_VALUE)
---------------------
                    8
                    9
                    2
```

如果带上这个子查询，查询结果又如何呢？

```
SQL> SELECT count(*)
  2         FROM OWNER1.S_SEND A
  3         WHERE
```

```
  4         A.STATE = 0 AND A.NETWORK_ID in
            (select to_number(attr_value)
            from OWNER1.a_sms_attr where attr_type = 1) AND
            A.MSG_TYPE_ID not in (2, 22, 23, 24, 25, 27);
  COUNT(*)
----------
         0
```

我们发现这个查询的结果经常是 0，不过有时候会是 1800 左右。这说明这个业务模块可能是处理完毕后修改状态位的模块。大多数情况下，这个查询返回的结果集是 0，少数情况返回的结果集不到 2000 条记录。那么，另外一个过滤条件 SERV_ID 的过滤性又如何呢？

```
SQL> SELECT count(*)
  2         FROM OWNER1.S_SEND A
  3         WHERE
  4         a.serv_id>=101112894255 and
            a.serv_id<=101113864255;
  5
  COUNT(*)
----------
     14664
```

明显这个过滤条件的过滤性不如前面的 STATE+MSG_TYPE_ID 组合。再进一步思考，如果采取 STATE+MSG_TYPE_ID+SERV_ID 的组合作为过滤条件，效果会如何呢？

```
SQL> SELECT count(*)
  2         FROM OWNER1.S_SEND A
  3         WHERE
  4         A.STATE = 0 AND
            A.MSG_TYPE_ID not in (2, 22, 23, 24, 25, 27) and
            a.serv_id>=101112894255 and
            a.serv_id<=101113864255;
  5    6    7
  COUNT(*)
----------
       165
```

这个组合的效果要比前面的过滤性更好。因此，如果要优化这条 SQL，最为简单的方法是创建这三个条件的复合索引，那么下一步我们就要考虑这三个字段的先后顺序。如果单纯从这条 SQL 来考虑，由于 STATE 的条件是等于，因此将其作为索引的首字段具有比较好的效率，但第二个字段是使用 SERV_ID 还是 MSG_TYPE_ID 呢？我们可以通过如下查询来查看到底哪个条件的过滤性更强：

```
SQL> SELECT COUNT(*) FROM OWNER1.S_SEND A
  2    WHERE STATE=0 AND A.MSG_TYPE_ID not in (2, 22, 23, 24, 25, 27);

  COUNT(*)
----------
      2270

SQL> SELECT COUNT(*) FROM OWNER1.S_SEND A
```

```
  2    WHERE STATE=0 AND a.serv_id>=101112894255 and
  3                      a.serv_id<=101113864255;

  COUNT(*)
----------
     13266
```

针对上述结果，我们发现MSG_TYPE_ID的过滤性更好，因此它更适合作为第二索引字段。不过上面的分析都是基于本SQL的，如果在创建索引时还要兼顾其他的SQL，那么设计索引的方式可能就会有所不同，对于这条SQL来说，虽然采用另外一种方式创建的包含这三个字段的复合索引可能不是最佳的，但是对于整个系统来说，就可能是最好的。

上述案例介绍了最简单的一种SQL优化方式，在多数情况下，我们很难让开发商去修改应用，因此索引的优化在SQL优化工作中尤为重要。在分析时，我们发现了主表（A表）上的索引范围扫描不是特别合理，并且存在很多过滤条件，因此我们会考虑是否换一个索引来提高这一层的扫描效率。而索引的创建是与其过滤效率有关的，索引的过滤效率不能仅仅从字段的选择性上来考虑，应该针对具体问题进行相应分析，以当前SQL作为前提条件，来判断哪些字段的组合才是最佳的。在进行这类SQL优化时，索引字段的选择性分析就十分重要了。并不是传统意义上的某个字段的值域越广，选择性就越强。对于某些具体的情况，需要根据业务的特点进行分析，这样得到的结论才更为准确。从值域以及全表范围来看，SERV_ID的选择性肯定好于STATE，但是在本SQL中，对于SERV_ID仅仅是一个范围扫描，该范围扫描的结果集远大于STATE=0的过滤结果，因此在这种情况下，我们认为针对本SQL，STATE字段的选择性优于SERV_ID字段。

在进行这类分析时，如果SQL中存在绑定变量会让我们很头痛，不过10*g*版本的dbms_xplan.display函数提供了一个渠道，可以帮助我们查看SQL进行绑定窥探时的绑定变量值，有了这个工具，在分析SQL时就不必担心绑定变量了。

另外要说明的是，本案例最终选择的添加索引的方法并不是最佳的解决方案，但是在某些优化场景下，最好的方案不一定是效果最佳的方案。这正是优化的魅力所在。

19.2 一个查找IP所属区域的SQL优化思路

前段时间，和QQ群里的一位朋友讨论了根据IP地址查找其所属区域的SQL语句。针对一个IP地址，如何才能找到其所属区域呢？IP地址区域表内包含区域编码、IP地址起始和IP地址终止3个关键字段。由于一个IP地址只能对应一个区域，所以这种查询返回的记录数为1或者0（0表示未找到该IP地址的区域）。

比如，IP地址表主要的3个字段是地区编码ANO、IP起始BNO和IP终止CNO。查找ANO的SQL语句如下：

```
SELECT ANO FROM TM1 WHERE BNO<=:IP AND CNO>=:IP;
```

为了找到一种比较好的解决方案，首先创建一个测试表。

```
drop table tm1;
create table tm1 (ano integer,bno varchar2(20),cno varchar2(20));
drop index idx_tm1;
create index idx_tm1 on tm1(bno  ,CNO);
declare
 va integer;
 vb varchar2(100);
 vc varchar2(100);
begin
 va:=0;
 for i in 1..300000 loop
  va:=va+1;
  select to_char(va,'099999999999') into vb from dual;
  select to_char(va+100,'099999999999') into vc from dual;
  insert into tm1 values(i,trim(vb),trim(vc));
  va:=va+100;
 end loop;
 commit;
end;
/
```

这样，我们就创建了一个具有 30 万条记录的测试表（实际环境中，可能存在上千万条记录）。接下来，我们来看看如下语句。

```
select /*+ index(tm1 idx_tm1) */ ano from tm1 where bno<='000000000015' and
cno>='000000000015';
Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=313 Bytes=11581)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TM1' (Cost=3 Card=313 Bytes=11581)
   2    1   INDEX (RANGE SCAN) OF 'IDX_TM1' (NON-UNIQUE) (Cost=2 Card=56)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        398  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

其中，consistent gets 是 5，看来性能还不错，接下来，换个参数试试，找一个比较大的值来测试。

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         62  consistent gets
```

```
          0  physical reads
          0  redo size
        399  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

我们看到，执行开销中一致读的数量变大了，实际上，随着参数值的增加，一致读的数量也会增加。

```
select /*+ index(tm1 idx_tm1) */ ano from tm1 where bno<='000030299900' and
cno>='000030299900';
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       3195  consistent gets
          0  physical reads
          0  redo size
        398  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

这是什么原因呢？先来看看这条 SQL 语句的 WHERE 条件，由于 WHERE 条件中使用了>=、<=这样的操作，因此对于这条 SQL 来说，需要进行索引的范围扫描。范围扫描会从第一个符合 bno<='000030299900'条件的索引项开始扫描索引，然后根据 cno 的值判断这个索引项是否满足条件，如果不满足就继续沿着叶节点链向后搜索。因此 IP 参数的值越大，扫描的索引叶节点数量就越多，相应地，一致读的数量也就越多。

实际上，在这个业务中，最多只存在一条符合条件的记录，而且 BNO、CNO 描述的区域是不重合的，因为符合 bno<=:v 条件的最大 BNO 值的记录才有可能满足 cno>=:v 条件。因此，如果顺着索引，按照 BNO 字段的值从大到小进行扫描，扫描到的第一条符合 BNO<=:V 条件的记录，如果也能同时满足 CNO>=:V 条件，就说明已经找到了记录；但如果这条记录不满足条件，那么在这张表中就找不到合适的记录了，最终的返回记录数为 0。

我们可以按照以下方式调整索引。

```
drop index idx_tm1;
create index idx_tm1 on tm1(bno desc,cno asc);
```

根据语义修改 SQL，添加 ROWNUM<=1 的条件。

```
select /*+ index(tm1 idx_tm1) */ ano from tm1 where bno<='000030299900' and
cno>='000030299900' and  rownum<=1;
```

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        398  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

调整后的语句如下：

```
select /*+ index(tm1 idx_tm1) */ ano from tm1 where bno<='000000546815' and
cno>='000000546815' and  rownum<=1;
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        399  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
select /*+ index(tm1 idx_tm1) */ ano from tm1 where bno<='000000000015' and
cno>='000000000015' and rownum<=1;

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        398  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

可以看出，开销明显下降了。当然，对于没有找到记录的参数，其开销是不会变化的，因为要对索引进行全扫描：

```
select /*+ index(tm1 idx_tm1) */ ano from tm1 where bno<='900000000015' and
cno>='900000000015' and rownum<=1;
```

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       3263  consistent gets
          0  physical reads
          0  redo size
        243  bytes sent via SQL*Net to client
        372  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

如果要对这种情况做进一步的优化，实际上只需找到第一条符合 BNO>=:V 条件的记录，取出其 CNO 和 ANO，然后判断 CNO 是否满足条件，如果满足就返回 ANO，否则返回 0。这样对于没有找到数据的情况，也可以达到最优的效果。

实际上，这个案例是根据索引扫描的特点，以及 IP 地址归属地查询中地址区域唯一性的业务规则来进行的。因此，在进行 SQL 优化时，仅仅依靠数据库技术是不够的，我们还必须在 SQL 中充分体现出业务的特点，这样可能才能达到最好的效果。

结 束 语

这本书是伴随着痛苦和快乐完成的。快乐是因为我又可以和大家一起共享知识和经验了。其实，整个写作过程也是我对知识的梳理过程，通过写作，我整理了自己处理过的上百个案例，并从中挑选了几十个案例，与大家分享。

痛苦是因为这两年是我们最忙的时期，而年过四旬的我已经没有了当年的那种激情。忙碌了一天之后，回到家甚至连电脑都不愿意打开。而几年前出版的《Oracle 优化日记》和《ORACLE RAC 日记》，基本上都是每天晚上一边在 QQ 群里和大家聊天，一边完成的。在这里，要特别感谢本书的编辑王军花女士，如果不是她一遍又一遍地催促，这本书不知道会写到猴年马月了。另外，老储的加入让我轻松不少，最费神的几个章节都交给他了，他对数据块和 ASM 结构的了解远在我之上，所以这些 INTERNAL 的解密交给他也算是众望所归了。这部分内容的详细程度让我都感到有些吃惊，解密做得如此彻底，看过这两节的内容，有心编写 DUL 的朋友可能又会蠢蠢欲动了。

老白在这本书中想要探讨的主要内容其实不是技术，也不是方法，而是一种理念：唯有理解了 Oracle 的本质，用 Oracle 的思维去考虑问题，才是 DBA 奋斗的目标。只有 Thinking in Oracle，才能在各种复杂的环境中找到问题的关键，并用正确的方法去解决问题。一切雕虫小技在这种境界面前，都是那么地渺小和苍白。

其实想做到 Thinking in Oracle 并不困难，只要用心去做每一件事，每个案例做完后都认真地总结一下，体会其中的 Oracle 本质，那么也许三五年，也许十年，你终归会达到高手的境界。

白鳝
第二次审定书稿时有感而发
2012 年 5 月 7 日
《DBA 的思想天空》
2012 年 2 月 20 日初稿完成于深圳
2012 年 3 月 20 日第一稿审订于巴厘岛 UBUD
2012 年 5 月 7 日第二稿审订于深圳
2012 年 6 月 4 日第三稿审订于深圳

DBA的思想天空

——感悟Oracle数据库本质

数据库的性能优化一直是DBA日常工作中非常重要的组成部分，然而很多DBA在学习了大量技术，参加了大量培训后，仍然会在实际工作中遇到难以下手的问题。实际上，在数据库优化工作中，方法和思路远比技术实现重要得多。

本书重在介绍Oracle数据库的性能调优方法及相应的工作思路，但并不拘泥于技术细节。作者通过大量真实案例，深度剖析了相关技术原理，同时还阐述了理论知识在实践中的应用方法。优化工作的本质其实就是透过表象探寻根源，解决问题实现调优，正所谓“思路是道，操作方法是技”，得道是极大的提升，也是DBA的思想精髓。

封面设计：ChoC 新浪微博：@出壳工作室 chocdesign@163.com
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈／投稿／推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机／数据库／Oracle

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-29443-2
定价：89.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。